Universidade do Minho
Escola de Engenharia

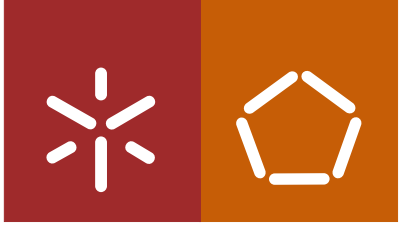
Bruno Silvestre Medeiros

A framework for heterogeneous many-core machines

January 2019

Bruno Silvestre Medeiros **A framework for heterogeneous many-core machines** UMinho | 2019

FCT Fundação para a Ciência e a Tecnologia
MINISTÉRIO DA EDUCAÇÃO E CIÊNCIA
COMPETE PROGRAMA OPERACIONAL FACTORES DE COMPETITIVIDADE
POPH PROGRAMA OPERACIONAL POTENCIAL HUMANO
QREN QUADRO DE REFERÊNCIA ESTRATÉGICO NACIONAL PORTUGAL 2007.2013



Universidade do Minho
Escola de Engenharia

Bruno Silvestre Medeiros

A framework for heterogeneous many-core machines

Doctoral Program in Informatics

Supervisor:
Professor João Luís Ferreira Sobral

January 2019

STATEMENT OF INTEGRITY

I hereby declare having conducted my thesis with integrity. I confirm that I have not used plagiarism or any form of falsification of results in the process of the thesis elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

University of Minho, 29 de Janeiro de 2019

Full name: Bruno Sebastião Mendes

Signature: Bruno Mendes

Acknowledgments

First, and foremost, I want to thank my supervisor Professor João Luís Sobral for supervising me during this long project and for his availability, all his insightful suggestions, and wiliness to do meetings and presentations on weakly bases.

I would like to show my appreciation towards full Professor Alberto Proença for the opportunity to do an internship at the University of Texas (Austin) to work with a fantastic HPC group led by Professor Keshav Pingali. A special thanks to Professor António Pina for the opportunity of lecturing the course of Computational Systems and for the CPD talks. I also want to thank Professor Mário Martins and full Professor João Fernandes for allowing me to contribute to the course of laboratory of Informatics III. I would like mention to the Engineer Albano Serrano for his availability to meet some of my requests regarding the cluster used on this work.

I want to take this opportunity to appreciate all the conversations with Rui Silva and Professor António Esteves about work and life in general, and Filipe Liu for all the coffee talks about the universe, science, politics, and HPC. I would like to express my appreciation to Rui Gonçalves for all his insightful suggestions to improve my work and also our technical discussions and heated discussions about economics and politics.

I also want to express my gratitude to my family and friends for all the emotional support, and a special thanks to my girlfriend Rafaela Conceição, not only for her emotional support but also for all her suggestions to improve my writing, and consequent contributions to this thesis. I could not forget to mention Pantera for staying up at night with me during the tight deadlines. Finally, last but definitely not least I want to dedicate this work to my parents and to the memory of my Aunt Paula and grandparents, José and Maria.

Bruno Silvestre Medeiros

Braga, January 2019

This work was supported by FCT—Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) grant SFRH/BD/82495/2011, and by ERDF—European Regional Development Fund through the COMPETE Programme (operational programme for competitiveness) and by National Funds through the FCT within project GAsPar PTDC/EIA-EIA/108937/2008.

A framework for heterogeneous many-core machines

Abstract

Software development is known for being a complex task, especially when parallelism is involved. This complexity can, however, be reduced by dividing the software into smaller manageable modules. This philosophy is embraced by modular programming, which promotes the *separation of concerns* in well-defined modules. Unfortunately, traditional parallel programming models (*e.g.*, OpenMP and MPI) are typically non-modular, leading to the mix of parallelism- and domain- related concerns. To aim for maximum performance the parallel applications should be tuned to the characteristics of the target architecture(s). However, in traditional approaches, this tuning process leads to unceasing and invasive adjustments of the domain code since the parallelism-related concerns are mixed directly in the domain code. This lack of modularity increases the complexity of parallel programming and jeopardizes the application maintenance. These problems are even more exacerbated in hybrid parallelism (*i.e.*, combining shared and distributed memory), which aims to exploit hierarchical systems such as clusters of multicore machines. Hence, these hybrid systems increase the complexity of the development of parallel applications even more, and consequently, emphasize the need for modular approaches.

This thesis exploits the notion that modularity, pluggability (*i.e.*, the ability to (un)plug modules without modifying the base code), and composability are key properties to make the process of developing parallel applications less complex. The first step towards achieving these properties is the separation of the parallelism-related concerns from the domain concerns and consequent encapsulation in proper modules. This thesis exploited the use of aspect-oriented programming (AOP) to achieve the separation of parallelism-related concerns and combined it with a methodology based on structured programming and design rules (*i.e.*, designing the domain code accordingly). The result is an aspect-oriented framework that enables the development of modular parallel applications. This framework intrinsically supports the development of applications with hybrid parallelism by composing, in a non-invasive fashion, several parallelism-related modules with a given domain code. This framework shines by combining the efficiency and expressiveness of popular HPC parallel programming models with the modular features of aspect- and object- oriented (OO) design.

As a result of studying AOP in the context of parallelism, we introduce the idea of *parallelism layers*, which combines the simplicity of well-known OO concepts (*i.e.*, class extension and method overriding) with the flexibility of AOP. On the one hand, this combination enables the users of our framework to add parallelism to domain code, using familiar concepts analogous to class extension and method overriding but without the limitations of OO inheritance. On the other hand, programmers can exploit the advanced features of AOP, which, among others, are helpful to extend the functionality of the framework. Hence, *parallelism layers* provide a simple yet flexible approach for the development of parallel applications. Finally, to reduce the complexity of parallel programming even further, we enhanced the *parallelism layers* with a methodology and a workflow to parallelize applications – including hybrid parallelizations – in an incremental and structured manner.

We evaluated the performance and programmability of our framework in comparison to other approaches by using a set of case studies and executing them in a cluster of multicores. We illustrated, using our framework and workflow, the entire process of developing efficient and modular parallelizations – from the sequential up to the hybrid version. Moreover, we show that our framework and workflow help to find more efficient parallelizations than the ones initially implemented. These results showed that *parallelism layers* are ideal for the quick prototyping and testing of different parallel strategies.

The results show that the parallelizations developed with the framework had a performance comparable to the intrusive parallelizations and, at the same time, were less verbose. With our approach, all the hybrid versions were seamlessly implemented. These hybrids were always faster than the correspondent versions that only used MPI processes, which emphasizes the potentiality of hybrid parallelizations in clusters of multicores.

Framework para máquinas heterogéneas com múltiplos núcleos

Resumo

O desenvolvimento de *software* é conhecido por ser uma tarefa complexa, especialmente quando o software pretende suportar paralelismo. No entanto, esta complexidade pode ser reduzida através da divisão do *software* em módulos mais pequenos e fáceis de gerir. Esta filosofia é adotada pela programação modular, que promove a *separação de facetas* em módulos bem definidos. Infelizmente, os modelos tradicionais de programação paralela (*e.g.*, OpenMP e MPI) não promovem o desenvolvimento de software modular, levando à mistura de facetas referentes ao paralelismo e ao domínio. Para tentar atingir o máximo de desempenho, as aplicações paralelas devem ser afinadas de acordo com as características da(s) arquitetura(s) alvo. No entanto, em abordagens tradicionais, este processo de afinamento leva a ajustes constantes e invasivos do código do domínio, uma vez que o paralelismo está diretamente misturado com este. Esta falta de modularidade aumenta a complexidade da programação paralela e dificulta a manutenção da aplicação. Estes problemas são ainda mais veementes em paralelismo híbrido (*i.e.*, a combinação de memória partilhada com memória distribuída), que tem como objectivo explorar sistemas hierárquicos tais como *clusters* de máquinas multinúcleo. Assim, estes sistemas híbridos aumentam, ainda mais, a complexidade do desenvolvimento de aplicações paralelas, e consequentemente, realçam a importância de abordagens modulares.

Esta tese explora a ideia de que modularidade, *plugabilidade* (*i.e.*, a capacidade de (des)conectar módulos sem modificar o código base), e composabilidade são propriedades fundamentais para que o processo de desenvolvimento de aplicações paralelas seja menos complexo. O primeiro passo para atingir estas propriedades é a separação da faceta referente ao paralelismo da faceta do domínio e o consequente encapsulamento de cada faceta em módulos apropriados. Esta tese explorou o uso de programação orientada aos aspectos (POA) para promover a separação de facetas referentes ao paralelismo e combinou-a com uma metodologia baseada em programação estruturada e regras de desenho (*i.e.*, o desenho do código do domínio de forma apropriada). O resultado é uma *framework* orientada aos aspectos que permite o desenvolvimento de aplicações paralelas modulares. Esta *framework* suporta intrinsecamente o desenvolvimento de aplicações com paralelismo híbrido através da composição, de forma não

invasiva, de vários módulos de paralelismo com um dado código do domínio. Esta *framework* sobressai por combinar a eficiência e expressividade dos modelos de programação paralelas populares em HPC com as propriedades modulares de programação orientada aos aspectos e orientada aos objetos (OO).

O estudo realizado nesta tese do uso da POA no contexto do paralelismo conduziu à introdução da ideia de *camadas de paralelismo*, que combina a simplicidade de conceitos OO bem conhecidos (*i.e.*, extensão de classes e reescrita de métodos) com a flexibilidade de POA. Por um lado, esta combinação permite aos utilizadores da *framework* adicionar paralelismo utilizando conceitos familiares, e análogos, à extensão de classes e reescrita de métodos, mas sem as limitações de herança OO. Por outro lado, os programadores podem tirar partido das funcionalidades avançadas de POA, que entre outras, são úteis para estender a funcionalidade da *framework*. Assim, *camadas de paralelismo* oferecem uma abordagem simples, mas flexível, para o desenvolvimento de aplicações paralelas. Finalmente, para reduzir ainda mais a complexidade de programação paralela, enriqueceu-se as *camadas de paralelismo* com uma metodologia e um fluxo de trabalho para paralelizar aplicações – incluindo paralelizações híbridas – de forma incremental e estruturada.

O desempenho e programabilidade da *framework* foi avaliado comparando-a com outras abordagens, usando um conjunto de casos de estudo que foram executados num *cluster* de máquinas multinúcleo. Mostrou-se usando a framework e fluxo de trabalho, o processo completo de desenvolvimento de paralelizações modulares e eficientes – da versão sequencial até à híbrida. Para além disso, mostrou-se que a framework e fluxo de trabalho ajudaram a encontrar paralelizações mais rápidas do que aquelas desenvolvidas inicialmente. Estes resultados mostram que *camadas de paralelismo* são ideais para a rápida prototipagem e teste de diferentes estratégias de paralelização.

Os resultados mostram que as paralelizações desenvolvidas com a *framework* obtiveram um desempenho comparável às paralelizações intrusivas, e ao mesmo tempo com menos verbosidade. Com a abordagem proposta todas as versões híbridas foram facilmente implementadas. Estas versões híbridas foram sempre mais rápidas do que as versões correspondentes que só utilizaram processos MPI, facto que realça a potencialidade das paralelizações híbridas em *clusters* de máquinas multinúcleo.

Contents

List of Figures

List of Tables

Abbreviations

AOP	**A**spect-**O**riented **P**rogamming
API	**A**pplication **P**rogramming **I**nterface
CCC	**C**ross **C**utting **C**oncerns
CPU	**C**entral **P**rocessing **U**nit
DM	**D**istributed **M**emory
DRC	**D**omain-**R**elated **C**oncerns
DRT	**D**ata-**R**elated **T**ransformations
DSL	**D**omain-**S**pecific **L**anguages
DSM	**D**istributed **S**hared **M**emory
GPU	**G**raphic **P**rocessing **U**nit
ILP	**I**nstruction **L**evel **P**arallelism
ITD	**I**nter-**T**ype **D**eclaration
JGF	**J**ava **G**rande **F**orum
JNI	**J**ava **N**ative **I**nterface
NUMA	**N**on **U**niform **M**emory **A**ccess
MD	**M**olecular **D**ynamics
MM	**M**atrix **M**ultiplication
MPI	**M**essage **P**assing **I**nterface
MPMD	**M**ultiple **P**rograms **M**ultiple **D**ata
OpenCL	**O**pen **C**omputing **L**anguage
OpenMP	**O**pen **M**ulti-**P**rocessing
OOP	**O**bject-**O**riented **P**rogramming
PP	**P**arallel **P**rogramming
PRC	**P**arallel-**R**elated **C**oncerns
RMA	**R**emote **M**emory **A**ccess
SM	**S**hared **M**emory
SMP	**S**ymmetric **M**ulti**P**rocessing
SPMD	**S**ingle **P**rogram **M**ultiple **D**ata

Chapter 1

Introduction

Ever since the beginning of the computer history, from the vacuum tube computers to the present day, that computer systems have increased in performance and capacity. For the past half century, the number of transistors has been growing significantly. To keep up with the performance demands, in addition to increasing the clock rate, hardware designers have also developed several other techniques, such as superscalar execution, branch prediction and speculative execution to name a few. However, these techniques eventually hit their peak, and end up reaching a point of diminishing returns [Sta12]. One of the primary sources of performance growth was the increase in clock rate. Unfortunately, this increase made processors reach speeds that led to power consumption and heat dissipation problems, among others.

Packing multiple processing cores into a single chip (*i.e.*, multicore) arose as an approach that could effectively exploit the growing number of transistors while keeping a sustainable balance between performance and power consumption. The goal of this hardware design is to achieve a higher overall system performance by enabling the system to run more tasks simultaneously. Instead of increasing the clock rate, this design increases the number of processing units *per* chip to improve performance, circumventing the heat dissipation issues [Sta12].

Naturally, with the increase in system performance, applications could become more complex and demanding. Parallel programming (PP) started to be used extensively to meet the demands of high-performance computing since it allows the division of tasks into smaller tasks that can be executed in parallel (a process known as parallelization). The performance gains depend on how much work is efficiently performed in parallel; therefore, these gains are highly dependent on how algorithms are designed and tuned. In PP, programmers are responsible for the application parallelization and ensuring the correct communication/synchronization among sub-tasks, which can be a hard task since it introduces several new potential bugs (*e.g.*, race conditions). Moreover, from the performance point of view, the programmer is also responsible for tuning the parallelization according to the characteristics of the target architecture to efficiently use underlying compute resources.

There are two broad classes of PP paradigms: shared memory (SM) and distributed memory (DM), where OpenMP and MPI are the most relevant standards for each PP paradigm, respectively. With the improvements made on multicores, it is unsurprising that clusters are evolving to clusters of multicores. Hence, it is expected that the number of applications using hybrid parallelization (*e.g.*, SM + DM) will increase in the future. Combining MPI with OpenMP is a commonly used hybrid strategy to exploit clusters of multicore machines. However, efficiently utilizing those clusters adds new challenges to software design.

1.1 Motivation

The shift towards multicore systems has pushed PP into mainstream programming. Moreover, the use of multicores in computer clusters requires a change in the way that software is developed and maintained. Software developers who aim to extract the maximum performance of the newer parallel architectures are not only compelled to choose competitive parallelism exploitation strategies for each of these architectures but also to further tune their software to the particularities of the target architecture. High performance is hard to achieve with a single base version, so programmers might have to develop multiple versions. These versions contain a lot of code replication among them, and only a portion of it is specific to a given platform. The fact that this interdependence between hardware and parallel concerns is expressed directly in the base code means that the programmers will have to unceasingly and invasively adjust their applications to keep up with architecture evolution and new performance requirements.

Developing high-performance parallel applications is an extremely complex task, not only because high-performance is not easy to achieve but also because current platforms combine multiple levels of parallelism. Hybrid systems, such as clusters of multicore machines, add an extra layer of complexity. In these systems, developers have to decompose the parallel tasks/data hierarchically. Moreover, when multiple paradigms are combined (*e.g.*, SM + DM), developers have to deal with the complexity and potential side effects of mixing parallel languages, in addition to dealing with the peculiarities of each one.

Modularity, pluggability, and composability are central properties to achieve, simultaneously, high-performance and manage the complexity of PP. Parallelism-related concerns (PRC) from different PP paradigms should be encapsulated into separate modules (modularity) that can be composed together (composability) to provide hybrid solutions – while keeping the domain code oblivious to this process (pluggability). To reduce the complexity of PP, programmers should first develop the domain-related concerns (DRC) independently from the PRC and only after should the PRC be (incrementally) added. The parallelism itself should follow a similar approach; programmers start with a simple parallelism exploitation strategy and then build sophisticated ones as needed by replacing, tuning and composing PRC modules.

Object-oriented (OO) programming languages, such as Java, materialize one of the most currently used software design philosophies, OO modular programming. However, OO falls short of modularizing crosscutting concerns (CCC), and unfortunately, PRC are known for being CCC [Har06]. Thus, it is common to mix them up with domain application concerns, jeopardizing the application maintenance and evolution. The situation is even more grotesque in some low-level PP codes where performance is the primary, and practically the exclusive goal. This mix of concerns threatens modularity and consequently composability.

Most PP standards (*e.g.*, OpenMP and MPI) provide appropriate abstractions to deal with the parallelism requirements of their programming model. To take advantage of hybrid environments, one might have to program with different PP languages and combine them. That not only increases the complexity of software design and the likelihood of bugs but also leads to even more code tangling and scattering issues. The parallelization of legacy code using such an approach will, most likely, result in a considerable amount of code rewriting.

Influenced by research done on the modularization and composition of concurrency concerns [Ber94, Lop97], Aspect-Oriented Programming (AOP) [Kic96] aims to solve the issues related to CCC. With AOP it is possible to encapsulate the CCC into units similar to classes, and later on, apply the CCC behavior to the domain code in a non-invasive fashion. AOP can, therefore, be used in conjunction with OO to complement it with an extra layer of transversal modularization. Previous work has explored the AOP concepts to modularize PRC [Har06, Sob06, ABVM10, PSR13, CV13, DM14, CL15]. To the best of our knowledge, none has provided an in-depth study of AOP and PRC that ended with a complete aspect-oriented Java framework (including a programming methodology) able to compose and modularize SM and DM PRC.

1.2 Goals

The non-modular approach for the development of parallel applications, imposed by traditional PP approaches, increases the complexity of PP and impacts software evolution and maintenance. Typically, traditional PP approaches focus exclusively on performance gains, ignoring, or even sacrificing, several other fundamental software qualities (*e.g.*, modularity). Moreover, in these approaches, PRC from different PP paradigms and DRC are tightly mixed, making it hard to reason about any of these concerns. In some applications, PRC are even treated as the dominant concern over the DRC. This thesis aims to deal with the previously mentioned problems by:

1. Reducing the complexity of PP through the development of a modular approach capable of developing efficient parallel applications. We believe that modularity, pluggability, and composability are fundamental properties for reducing the complexity of PP. Separation of concerns (*e.g.*, separating domain- and parallelism- related concerns) is the first big step towards developing a system with those properties. We aim to exploit the potential of AOP regarding the separation, encapsulation, and composition of concerns and to understand its impact in the context of parallelism;

2. Developing an aspect-oriented framework that combines the expressiveness and efficiency of traditional PP models (*i.e.*, OpenMP and MPI) with the support of modularity, pluggability, and composability properties. This framework aims to promote a more structured, modular and less complex development of parallel applications. The programmer should be able to compose seamlessly different parallelism-related modules with the domain code. We argue that a framework with these characteristics enables the quick testing, tuning, and development of different parallel strategies and facilitates the composition of multiple levels of parallelism (*i.e.*, hybrid parallelizations);

3. Validating our approach with a set of case studies, which include legacy code and tuned code. The validation is performed by comparing the programmability and efficiency of the framework with traditional approaches;

4. Providing methodologies and strategies that take advantage of AOP, to guide the development of parallel applications using our framework and to attenuate (potential) weaknesses of AOP.

1.3 Outline

Chapter 2 presents the main parallel architectures and programming models relevant to this work. It describes SM and DM programming models, namely multithreading and message-passing, and their most popular standards, OpenMP and MPI, respectively. Additionally, it touches on Java support for parallelism and concurrency. Moreover, it presents hybrid parallelizations, with the main focus on the combination of MPI with OpenMP. Finally, this chapter concludes with performance considerations for achieving high-performance parallel applications.

Chapter 3 starts with the AOP background, its main motivations and the use cases where it is commonly employed. It illustrates the problems of mixing the PRC with DRC, using a hybrid parallelization as an illustrative example. It describes, in some detail, the concepts and functionalities of AspectJ (*i.e.*, an AOP extension for Java) most relevant for the understanding of this work. It finishes with a description of strategies and patterns used to create reusable aspects, most of which also were used in our aspect framework.

Chapter 4 shows an overview of the related work. It describes relevant libraries, frameworks, and languages for SM, DM and hybrid parallelism. Moreover, it references OO mechanisms and skeletons frameworks. Finally, it revises the most relevant AOP work regarding PRC.

Chapter 5 presents the proposed approach and describes the requirements, conceptual overview and detail description of the AOP framework, concluding with a discussion about the work developed.

Chapter 6 shows the performance and programmability validation of our framework. It describes in great detail, for two case studies, our workflow for developing efficient and modular parallelizations, starting with their (sequential) domain code and finishing with their hybrid parallelizations. Finally, it shows the validation work done for a set of case studies from the well-known *Java Grande benchmark suite* (JGF) [SBO01].

Chapter 7 draws conclusions about the work displayed in this thesis, showcases the main contributions and presents potential researching paths for future work.

Chapter 2

Parallel architectures, programming models and languages

2.1 Shared Memory

In a SM hardware, also known as tightly-coupled, processors can access a global shared memory, via some interconnection network (*e.g.*, a shared bus), which can be used by the processors to communicate with each other through memory reads and writes. Usually, in the literature, SM architectures are categorized according to the time that processors take to access the global shared memory [Sta12] or according to their memory organization [ERAEB05, DSK05]. Based on the processors' global memory accessing time SM architectures can then be divided into either UMA (uniform memory access) or NUMA (non-uniform memory access).

2.1.1 Hardware

2.1.1.1 UMA

Symmetric multiprocessing (SMP) is one of the most common types of SM systems. SMP has a common centralized memory that takes the same time to be accessed by processors, regardless. Hence, this design falls into the UMA category. In fact, in the literature, SMP is sometimes used interchangeably with UMA (*e.g.*, [ERAEB05, EGCSY03]) or used to categorize a UMA system with a bus-based interconnect network (*e.g.*, [Ale00, Tan01, Vin99]).

In a UMA system the processors are usually connected to the memory modules through a shared bus [Ale00] or a switching network (*e.g.*, crossbar [SM07] or multistage [LSB88]). The former scheme scales the least, but it is the most flexible (*i.e.*, easy to attach more processors to the bus) and the simpler of the two [Sta12]. The latter scheme is regarded as consuming more energy [Pad11], being more complex and still not scalable enough [DSK05].

The bus organization suffers from scalability issues because it is shared among processors and contention will eventually occur when multiple processors try to access the main memory [Tan01]. An increase in the number of processors increases the likelihood of processors blocking when accessing the main memory, waiting for the bus to become idle. This factor reduces the scalability of the bus organization and makes it dependent on the bandwidth of the bus [DSK05].

The use of cache memory can increase the scalability of SM systems, but it requires the use of protocols. These protocols use the interconnection network to perform its routines (*e.g.*, dealing with cache misses). Increases in the number of processors will likely increase the overall cache misses and lead to higher traffic passing through the interconnection network [BDGS92].

2.1.1.2 NUMA

The issues underlying the interconnection network, the cache coherency protocols and the single centralized SM limit the scalability of UMA systems – NUMA architecture [GTM96] was designed to solve these limitations. In the literature, NUMA is categorized as being either a distributed shared memory (DSM) architecture [DSK05, EGCSY03], specifically a hardware-based approach of DSM [Vin99], or a SM architecture [Sta12, PH13]. Some divide NUMA further into CC-NUMA [Sta12, DSK05] if it uses cache coherency or NC-NUMA [Tan01] otherwise. This work covers only CC-NUMA (henceforth simply NUMA), and treat it as a SM architecture.

In NUMA, in contrast to UMA, processors do not access a unique centralized memory. Instead, processors are arranged in groups (*i.e.*, NUMA nodes), each with its main memory. The NUMA nodes are connected through a high-speed interconnector and processors within those nodes access their local main memory using an interconnection network. This design relieves the contention and memory bandwidth issues of UMA and, consequently, increases scalability.

Since NUMA nodes are connected processors can access the main memory of their local node and the main memory of the other nodes (designated remote memory). In contrast to UMA, where all processors spend about the same amount of time to access memory, in NUMA it is slower to access remote memory than local memory. Hence, the performance in NUMA is more dependent on data allocations than in UMA [DSK05].

Most NUMA architectures use cache memory attached to each processor. However, cache coherent protocols [Law98] in NUMA are typically more complex than UMA, because of the dimension (*e.g.*, more processors/caches) and the different accessing times between local and remote memory [Sch01]. NUMA architectures such as Cray T3D [CCS95] that do not use hardware cache coherent protocols rely on additional software to ensure cache coherence.

As with UMA, the scalability of NUMA will be, at some point, limited due to contention on the interface connecting processors to memory (further aggravated with the cache protocols), main memory bandwidth, and the penalty of remote memory access. These problems are, in fact, chronic to the SM architectures [DSK05].

2.1.1.3 Multicores

In the previous decades, hardware designers were able to utilize the ever increasing number of transistors to improve the overall performance of computer systems. At the time, the trend was to build single-core processors with higher clock frequency and equip them with sophisticated techniques to increase the number of instructions performed *per* cycle. These techniques focus mainly on increasing instruction-level parallelism (ILP) [Sta12]. It was an attractive trend for non-HPC programmers because an application could be designed without too much emphasis on performance and hopefully, it would execute faster as a result of hardware improvements.

At a certain point, the trend of increasing ILP and clock frequency was no longer scaling satisfactorily. The increase of clock frequency eventually led to power consumption and heat dissipation issues. Moreover, the strategies to increase ILP started to become prohibitively complex and giving diminishing returns. In addition to these issues, memory became a performance bottleneck because it did not improve at the same rate as the processor, making this trend unhelpful for memory-bound algorithms. The answer for this scalability sluggishness was to build more, but simpler, parallel processors [ONH+96] that would rely on explicit PP to scale further.

In the last decade, SM architectures have experienced a new trend entitled multicores, or chip multiprocessor [Sta12, Tan01], in which the SMP/NUMA design is applied at the core/chip level. There are some slight differences between multiprocessors and multicores, however, for simplicity reasons, one can think of multicores as a combination of *"two or more processors (called cores) on a single piece of silicon (called a die)"* [Sta12]. The operating system treats both cores/processors as the same. However, one relevant difference is that in multicores it is common to share not only the main memory among cores but also the last level of cache.

K. Olukotun *et al.* in [ONH+96] argued that multicores were inevitable due to two major forces, namely a technology push and an application pull. Regarding the technology, Olukotun foresaw that the superscalar execution model (a technique used to increase ILP) would eventually stagnate. The application argument was based on the study [Wal91] that divided parallelism in applications into two groups. Both groups would benefit performance-wise from a low latency inter-core communication chip with simpler cores that could achieve high clock rates.

Multicores offers lower latencies compared with the traditional multiprocessors because in the latter processors needed to go off-chip to communicate with each other. Moreover, multicores can exploit the lower latencies of communication on-chip by sharing the last level of cache among them. Hence, parallel applications can scale better in multicores than in multiprocessors, especially when these applications exploit fine-grained parallelism [ONH+96].

In the literature, the term processor is context sensitive and sometimes is used interchangeably with the term core [Sta12, PH13, Tan01]. Therefore, to avoid ambiguities, and henceforth, in the scope of this thesis, the terms core and processor will be used interchangeably. Consequently, the term SMP will be referring to symmetric multiprocessors/multicores.

2.1.2 Programming model

In simple terms, from a software perspective, SM can be defined as when parallel tasks can communicate with each other using shared variables. Although SM hardware and SM programming models appear to be tightly-coupled, in reality, SM programming models are an abstraction that can be applied to DM architectures as well (*e.g.*, DSM [Vin99]). This semblance of SM, in the DSM systems, can be implemented through additional software [BZS93, KCDZ94], by using the hardware itself [GTM96] or a combination of the two [MD09].

Multithreading is a common SM model used in machines with multicores and the focus of this thesis. In this model, multiple threads run in parallel and communicate with each other through shared variables. Besides sharing a heap, threads also have their (own) stack, ID, and private state. This model is generally used to divide a common bigger task into smaller ones (*e.g.*, loop iterations) that will be assigned to threads.

Programmers can create/manage threads and related features either by explicitly calling library routines [BFN13] or by declaring directives that will instruct the compiler to do so [DM98]. In both cases, the programmer is responsible for dealing with the issues arising from parallelizing an application with SM multithreading approach. Programming languages such as C/C++ have at their disposal a panoply of frameworks, libraries and language extensions for multithreading parallelization, such as Intel-TBB [Phe08], OpenMP [DM98], POXIS Threads [BFN13]. Some of these libraries (*e.g.*, OpenMP) provide higher-level mechanisms that deal with the lower-level details of multithreading parallelization (*e.g.*, thread creation) mitigating some of the programmers' efforts. With other libraries (*e.g.*, POXIS Threads), the programmer is responsible for both high and low-level specifics. The choice between these two types of libraries is a trade-off between less programming effort at the cost of expressiveness and flexibility [Bre09].

2.1.2.1 OpenMP

OpenMP is a standard to implement SM parallelism through compiler directives, library routines, and environment variables. With OpenMP programmers use language annotations, named *pragmas*, in strategic points of the source code, to *communicate* to the compiler that PRC (*e.g.*, parallel *for*) should be inserted in these points. OpenMP was released in 1997 and was initially intended for CPUs only. However, in 2013, with version 4.0 [apivb], the OpenMP standard introduced new directives especially aimed at coprocessors, which provide some of the features of OpenACC[1][CEtPG11]. The last official release of the OpenMP standard, before the expansion to the coprocessor, was 3.1 in 2011 [apiva]. This thesis covers in more depth this version, specifically the C/C++ sections because most of the features of the SM library of our framework are based on that standard.

[1] OpenACC follows a similar philosophy of OpenMP but applied to coprocessors.

The OpenMP parallelism is mainly specified through pragmas, as shown in Figure 2.1. The directive is applied to the block of code right below the pragma, which can be a single statement or multiple ones within brackets. At compile-time, the compiler reads the directives and generates the code necessary to produce the desired result. In the example of Figure 2.1, the compiler generates code to assign the *for* loop iterations to the threads.

```
1  #pragma omp parallel for
2  for(int i = 0; i < N; i++) {...}
```

Figure 2.1: Example of OpenMP annotations.

The OpenMP directives are only interpreted by the compiler if the code was compiled with a specific flag. This behavior allows for some programs to be executed in multi- and single- core machines without any code modifications. However, there are codes parallelized with OpenMP that due to their nature produce different results when running in parallel or sequential [EBFJ16].

OpenMP 3.1 focus on parallel regions, work-sharing constructors, tasks, synchronization mechanisms, and data environment directives. The execution model of OpenMP starts with a single thread named *initial* thread [apiva]. Whenever this thread encounters a parallel region, it requests a team of threads and becomes the master of that team. The master thread has the ID 0 while the remaining threads have IDs ranging from `1` to `N - 1`, where N is the total number of threads within the team. To each thread is assigned an implicit task, which represents the code enclosed by the parallel region. If nested parallelism is active and a thread of a team encounters a new parallel region, this thread repeats the process previously mentioned of requesting a new team. If nested parallelism is disabled, which it is by default due to the additional overhead that it introduces, then the inner parallel regions are ignored.

The creation of a team of threads to execute a given block of code *per se* is not very powerful. The user would have to explicitly divide and assign work to threads to reduce the application execution time. Therefore, OpenMP offers work-sharing constructors that are used (in conjunction with parallel regions) to divide work among threads. The OpenMP 3.1 standard for C/C++ defines three different work-sharing constructors, namely *single*, *sections* and loop. All of them plus the parallel region by default have an implicit barrier at the end of their execution. This barrier works as a synchronization point among threads of the same team. Nonetheless, the user can disable these implicit barriers by adding the clause *nowait* to the constructor annotation.

The *single* constructor restricts a block of code to be executed once, and only by the first thread that reaches it. The *sections* directive specifies, within a parallel region, multiple blocks of code that can be executed by different threads in the team. The loop constructor assigns blocks of iterations (called chunks) of a *for* loop, with a specific structure[2], to threads. In the loop constructor, the user can choose the chunk size as well as how should these chunks be

[2]The *for* loop structure imposed by the OpenMP standard can be found on page 40 of the OpenMP 3.1 standard document [apiva].

distributed among threads (*i.e.*, statically or dynamically). Moreover, it is possible to parallelize consecutive loops by using the *collapse* clause. The iterations of the collapsed loops are treated as a unique set of iterations that will be distributed among threads.

The loop constructor provides some predefined distributions, namely static, dynamic, guided, auto and runtime. The static, commonly used by default[3], divides the iterations into chunks and assigns them in a round-robin fashion using the thread ID in ascending order [apiva], whereas in the *dynamic* the assignment is performed on a first-serve basis. When the user does not define the chunk size its value is the corresponding default value according to the distribution chosen. For the dynamic, the default is one, whereas for the static is (approximately[4]) the number of iterations divided by the number of threads. The guided scheduling works similar to the dynamic, however, every time a chunk is assigned it decreases its size until it reaches a particular value, remaining the constant from that point forward. Finally, with auto, the compiler will choose how to distribute the loop iterations, while with runtime the scheduling scheme and chunk size are read from environment variables at runtime.

Dynamic scheduling offers, potentially, better distribution of work among threads than static scheduling, however, it also has a higher overhead. This additional overhead comes from the use of a locking mechanism during the assignment of work. Moreover, another advantage of the static over the dynamic scheduling is related to NUMA node locality. If two or more parallel loops have the same iteration range, chunk size and are working with the same data (*e.g.*, array), then threads will be working with the same data regions in-between loops. Hence, threads will most likely work with data that is already in their node. This fact is especially useful when applying the first-touch memory technique [Lam13].

OpenMP by itself does not solve data dependencies and race conditions, however, it offers synchronization constructors, explicit locks and data environment directives to do so. The synchronization constructors control the execution of threads in a team. For instance, the *master* directive restricts code to be only executed by the master thread. Other directives include *critical* region and explicit barriers. The former enforces mutual exclusion, meaning that only one thread at a time can execute a given section of code. The latter specifies a synchronization point, at which threads wait for all threads in the team to reach that point. Finally, the user can explicitly create and manage locks as well as use a set of atomic operations.

Data environment directives control the data scope inside parallel regions and work-sharing constructors. Data directives allow the user to instruct the compiler on how specific data, created before a given constructor, should be handled inside the scope of that constructor. Furthermore, data directives also allow to express how, and if, the data should be updated after the execution of the constructor. Data can be either shared among threads or private to each one of them.

[3]The default value can be changed using the internal control variable *def-sched-var*.

[4]If the blocks cannot have the same size, the difference between them should be, at most, one loop iteration.

For shared data, the most commonly applied clauses are *private*, *firstprivate*, *lastprivate* and *reduction*. The first clause creates a private copy of a variable *per* thread ignoring the initial value of that variable, the second and third clauses do the same as the first but dealing with the initial and final values of the variable before and after entering a given constructor region, respectively. Finally, the *reduction* clause creates a copy of a variable *per* thread and at the end of the constructor reduces these copies based on the chosen reduction function.

The programmer can change the visibility of variables by using the data environment directives. However, there are some restrictions over their usage. In the context of this thesis, the most important are:

- the *shared* clause can only be used in private data, whereas the *reduction* and *private*-related clauses can only be used in shared data;
- reference types must not be used in *shared* and *private*-related clauses;
- when an instance of a class is used in *private*-related clauses that class should have *"an accessible, unambiguous default constructor"* [apivb].

In OpenMP 3.1 only local variables of primitive data-type can be used in the *reduction* clause. However, in OpenMP 4.5 it is possible to reduce arrays (statically allocated) and to create user-defined reductions that allow the reduction of heterogeneous data-types as structures. Unfortunately, it is still not possible to reduce dynamically allocated data.

The OpenMP API allows users to explicitly call functions and set environment variables to be read at compile-time. Most of these functions fall under the set/get type category, such as setting/getting the number of threads within a team. However, not all functionality is exposed, for instance, users cannot access the thread pool directly. Nevertheless, the possibility to make explicit calls to the API provides useful flexibility.

From OpenMP 3.1 to 4.5 a panoply of new features was introduced, echoing the recent trend to support coprocessors and providing greater expressiveness. Some of these features target performance in multicore machines, for instance, thread affinity within a parallel region and the support for SIMD parallelism. Others such as specifying data dependencies, user-defined reductions, and the possibility of canceling a constructor during its execution – similar to the semantics of a break – target correctness. Additionally, the task constructor functionality was also extended to allow, among others, to turn the iterations of a loop into tasks to be queued.

A large amount of functionality was added from OpenMP 3.1 to 4.5, but the biggest *jump* was felt from 3.1 to 4.0, where the standard roughly doubled its number of constructors [Car]. With the most notable change being, arguably, the introduction of offloading capabilities tackling some of the goals of OpenACC[5].

[5]In 2013, year of the release of OpenMP 4.0, OpenACC released its 2.0 version.

2.1.2.2 Java language support for parallelism/concurrency

In Java, the class *Thread*, available since JDK 1.0, allows the creation of a new thread inside a given application. This class implements the functional interface named *Runnable* that requires the implementation of the *run* method, which provides the thread with the code to execute. The programmer can provide the concrete implementation of the *run* method either by creating a *Runnable* object and passing it to the *Thread* constructor or by extending the class *Thread* itself and overriding its *run* method. If the main goal is only to provide the thread with the code to be executed, and not to extend or alter its functionality, then it is preferable to pass the *Runnable* object instead of extending the class *Thread*. To create the *Runnable* object, which will be passed to the *Thread* object, the programmer can use one of the following commonly used approaches (sorted from the most to the least verbose): - creating a new class that implements the *Runnable* interface; - creating the *Runnable* as an anonymous class; - using lambda expressions (a feature introduced in Java 1.8).

With the increasing adoption of multithreading, Java evolved in the direction of providing higher-level abstractions on top of the class *Thread*. Creating threads is a computationally demanding operation [Oak14], therefore, in some scenarios, it might be desirable to reuse them. Hence, Java 1.5 added the possibility of creating different types of thread pools (*e.g.*, *newFixedThreadPool* and *newCachedThreadPool*), which can be further tuned to satisfy different requirements.

Java 1.7 introduced a new type of thread pool named *ForkJoinPool* that is suitable for divide-and-conquer type algorithms [Oak14]. This type of parallelism, typically, results in a large number of parallel tasks that are handled by a smaller number of threads [Oak14]. Moreover, often the completion of some of these tasks depends on other tasks results.

ForkJoinPool allows threads to suspend one task in favor of another task, a feature that does not exist in older thread pools. Furthermore, *ForkJoinPool* has a work stealing mechanism that enables threads to *steal* work from the queues of other threads. Naturally, the choice between the different types of thread pools depends on the specific needs of the application.

Besides the API to create threads and execute tasks in parallel, Java also provides concurrent features to deal with the correctness of multithreaded applications. These features include semaphores (*e.g.*, *Semaphore*), locks (*e.g.*, *ReentrantReadWriteLock*), concurrent collections (*e.g.*, *ConcurrentMap*), thread-safe variables with atomic operations (*e.g.*, *AtomicInteger*) and so on. Regarding concurrency, Java 1.8 has come with several new features [Ora18] and with the promise of parallelization of a few methods (*e.g.*, *Arrays.parallelSort*).

2.2 Distributed Memory

Distributed memory (DM) architectures, also known as message-passing systems [ERAEB05, Pad11] or loosely coupled, can be an alternative to SM hardware. In a DM architecture processors have their local private memory, instead of a shared one, and are connected among each other through some interface network. Processors can work independently, without the concern that changes to their memory may affect other processors. This hardware approach is commonly used in parallel supercomputers [UIT94, HOF+12] and clusters of machines [PH13]. Naturally, the overhead of communication among processors from different machines is higher than among processors from the same machine.

Parallel applications running in DM hardware usually use multiple processes, each mapped to a different processor. These parallel applications can be programmed using different approaches to take advantage of DM hardware, such as task-farming [CKPD99, ABG02], SPMD (*i.e.*, Single Program Multiple Data)[PB04, CD01] and MPMD (*i.e.*, Multi Programs Multiple Data)[CCvEK97]. In a DM system, each process can execute its (own) program or different execution paths of the same application. Specific interfaces (*e.g.*, sockets) are used to establish a communication channel among processes. The programmer is responsible for explicitly defining when and how the communication among processes should be performed [ERAEB05]. Message-passing is one of the most used models to exchange data between processes, with MPI [For94] being one of the most used standards [WPH03] and considered to be *de facto standard* for message-passing [GHD00, LSM11].

In a DM task-farming model, also known as the master/slave model, one process (master) controls and distributes work to the remaining processes (slaves). Usually, in this model, the master process decomposes a bigger task into smaller ones and assigns them to the slaves. The task distribution can be performed statically or dynamically, and usually, the exchange of messages occurs only between the master and the slaves [MEB99]. In the SPMD approach, each process runs an instance of the same application but work only with a code/data subsection of that application. In the MPMD approach, processes run different applications. This approach is commonly used in applications with irregular communication patterns and computations loads [CCvEK97].

2.2.1 MPI

MPI is the most used standard to implement the message-passing model [WPH03, GLS14, DMCN12] with its first version released in 1994 [For94]. The standard initially included the programming languages C, C++, and FORTRAN. However, after version 3.0, published in 2012 [For12], the C++ bindings were removed. Nevertheless, it is possible to find MPI implementations even for the languages not supported by the standard. For instance, OpenMPI [GFB+04],

one of the most popular MPI implementations [VGRS16], officially supports Java bindings. OpenMPI Java [VGRS16] relies on the Java Native Interface (JNI) to bind the Java and the C libraries and implements most of the MPI 3.1 standard.

The MPI 3.1 standard offers several features [GLS14, GHTL14] to build sophisticated and high-performance parallel applications, such as collective communication, the specification of process topologies, the grouping of processes and remote memory accesses (RMA) [HDT+15]. Although MPI provides several advanced features, as stated in [GLS14] the majority of the MPI programs can be implemented using a set of only six routines, namely: - MPI process initialization and termination; - getting the process ID and the number of processes; - the send and receive of messages between two processes. Nevertheless, the remaining features are useful to develop more robust, flexible and efficient parallel applications [GLS14].

All MPI processes created during the execution of an application are part of a global group. Nonetheless, the user can create additional groups with different sets of processes. This feature can be useful, for instance, to restrict communication to the processes belonging to a specific group. Each MPI process within a group has a unique ID (*i.e.*, *rank* in MPI terminology) that ranges from 0 to $\texttt{N} - 1$, where `N` is the total number of processes in the group.

MPI offers three types of communication, namely point-to-point, collective and one-sided. The first type refers to communications between two processes (*e.g.*, *send/recv*), the second type to communications that involve all processes in a group (*e.g.*, *all reduce*), and the third type to a process accessing the memory of another process through well-defined MPI operations (*e.g.*, *put*, *get* and *update*). In MPI the content of a message does not have to be a contiguous data structure. For non-contiguous data structures, the user can use the MPI *derived datatypes* and explicitly define the data layout, or can, alternatively, copy that data into a contiguous data structure and use it as the message.

To use point-to-point communication routines (*e.g.*, *MPI_Send* and *MPI_Recv*) it is necessary to specify, among several parameters (including the message and its size) the ranks of the sender/receiver processes. Nonetheless, MPI also allows a process to receive a message from unspecified processes. This option is particularly handy on the task-farming model, where the master might not know in advance the process that will request work next. MPI provides blocking and non-blocking versions of the point-to-point communications.

Collective communications provide a convenient way of communication among all processes within a group. Without this feature, for example, to send the same message from the master to all the other processes, every single pair of point-to-point communication between the master and the remaining processes would have to be explicitly defined. MPI provides collective routines to split/merge data structures (*e.g.*, *MPI_Scatter* and *MPI_Gather*), which are ideal for domain decomposition parallelism. Moreover, there are routines to perform global synchronization (*e.g.*, *MPI_Barrier*), to send the same data from one process to the remaining

processes (*e.g.*, *MPI_Bcast*) and to apply a collective operation on the transferred data (*e.g.*, *MPI_Allreduce* and *MPI_Scan*). The collective communications can be divided according to the number of processes that send and receive the messages, namely: - *one-to-all* (*e.g.*, *MPI_Bcast* and *MPI_Scatter*); - *all-to-one* (*e.g.*, *MPI_Reduce*); - *all-to-all* (*e.g.*, *MPI_Allgather*). For some of the collective communications, MPI provides also blocking and non-blocking versions of them.

Communicator is an important MPI concept that defines a communication space where a subset of processes can communicate with each other. The MPI standard defines two different categories of communicators, namely intra- and inter-communicators. The former defines communication among processes belonging to the same group and the latter between processes from different groups. In the context of this thesis we are only interested in intra-communicators, henceforth, named communicators. Communicators are composed by a group of processes, context[6] and (optionally) the virtual topology [GLS14]. The concept of different communication spaces helps to distinguish between message exchanges from different contexts. This concept allows the development of MPI applications/libraries that can work cooperatively at the same time, without the risk of their messages being intercepted by undesirable processes. Indeed, one of the goals of MPI was to ease the creation of libraries on top of MPI implementations [GLS14].

2.3 Hybrids

A cluster connecting several multicore machines (*i.e.*, nodes) allows the scaling of hardware to thousands of cores. Nowadays, these nodes are usually NUMA with multiple sockets that provide several cores. Two common strategies to develop applications scalable in those clusters are the use of a: - DM model only; - a hybrid solution that combines SM and DM models.

The simplest method to explore parallelism in a cluster of multicores is to use message-passing with one MPI process *per* core. Another approach is to use a hybrid of MPI with some SM library/extension (*e.g.*, OpenMP). Recently, with the introduction of SM capabilities in the MPI 3.0 standard, it is possible to find implementations (*e.g.*, [HDB+13]) where MPI is used to provide both the DM and SM models. For instance, using message-passing and the MPI SM programming API respectively for inter- and intra- node communication.

In this thesis we focus on one of the most common hybrid approaches, the combination of MPI with OpenMP [BM08, RHJ09, JJM+11, MLAV10, ZMWK17], henceforth, just designated as hybrids. The simplest way to implement hybrid parallelizations is to use OpenMP in specific hot spots of the code (*e.g.*, parallelizing computationally demanding loops) while the MPI routines are called sequentially either by just performing them outside the parallel regions or by restricting them to only one thread (*e.g.*, *master* or *single*). A second approach is to use threads, not only to parallelize chunks of code but also to overlap communication with computation [MLAV10].

[6] Meta-data generated by the MPI implementation to distinguish between different communicators [MCS00].

In such implementations, one or more threads perform the communications while the remaining threads execute computation in parallel. Naturally, there are advantages and disadvantage to both approaches. The first one is simpler to implement, easier to maintain and debug, whereas the second has the potential to scale more than the first.

Concerning MPI and multithreading, the standard offers four possible levels of thread support, however not every MPI implementation provides them[7]. To use MPI in a multithreaded application, the programmer has to explicitly notify the MPI library of the level of thread support that the hybrid parallelization will need. The supported levels are: - *single* (the default value), the application is not multithreaded; - *funneled*, the application is multithreaded but only the master thread does the MPI calls; - *serialized*, any thread can perform the MPI calls but not concurrently; - *multiple*, threads can concurrently make MPI calls. These levels establish a contract between the application and the MPI implementation so that the latter can work with a set of assumptions to provide a correct and efficient implementation. From all the levels the last one has the highest overhead since MPI has to take extra measures to ensure correctness.

A hybrid approach requires a greater programming effort than using simply MPI, so its use is only justified if it can increase performance. Some of the disadvantages of hybrids compared with the MPI only approach are: - requires a higher level of expertise [LW17], and is harder to implement, debug and maintain since programmers have to be knowledgeable about the SM and DM programming models; - the need for multithreading support by the MPI implementation [LW17]; - the introduction of additional overheads from the SM model. On the other hand, the advantages of the hybrid approach are: - the reduction in memory consumption; - the improvement of load balancing; - the reduction in messages exchanged; - being more suitable for fine-grained parallelism; - the fact that it conceptually mirrors the hardware hybrid structure.

Typically, a hybrid approach consumes less memory than the MPI only approach, because there is less data replication among processes, particularly in MPI parallelizations using the *Ghost cell pattern* [KS10], and less memory reserved for the MPI buffers. Moreover, threads use less memory than processes, especially in Java where each MPI process will execute an entire JVM. The works in [SJF+10] and [IJ11] present case studies with memory footprints that were considerably reduced after moving from MPI only to a hybrid approach.

Hybrids can improve the load balancing by exploiting the OpenMP dynamic scheduling [RHJ09]. Also, they can improve the load balancing of applications with processes that can have different workloads or/and that are executed in machines with different capabilities, by tuning the number of threads of each process accordingly. It is possible to implement dynamic scheduling in MPI, however, compared with OpenMP, its implementation requires a greater programming effort since by default it is not provided by the MPI standard, and (usually) with a higher overhead. The works [HT00, RHJ09] show the contribution of hybrids to load balancing.

[7] MPI standard compliance implementations are not obliged to provide multithreading support.

Besides improving memory consumption and load balancing, hybrids can also improve performance by reducing the exchange of messages and the number of processes participating in collective communication routines [LW17]. The performance of these collective routines is highly depended on the hardware and its MPI implementation. The reduction of the ratio of communication to computation leads to the increase in the granularity of the parallelism, and consequently, its scalability. Applications that are limited by the number of processes, that scale poorly in MPI but well in OpenMP, or that have two levels of parallelism (*e.g.*, domain decomposition among processes and threads) can also benefit from hybrids. Finally, hybrids can also help when hardware constraints restrict the scalability of a MPI parallelization, for instance, the latency and number of ports of the network, memory constraints of the machines and so on.

Hybrids can be implemented by starting with the MPI version and then adding OpenMP on it and *vice-versa*. For maximum performance, it is paramount to develop efficient MPI and OpenMP parallelizations before combining them. The mapping of processes to resources in a hybrid parallelization can vary depending on the algorithm and hardware topology. This mapping consists, usually, in one or more processes either *per* machine (with threads sharing the node memory) or *per* socket (with threads sharing the socket memory).

2.4 Performance considerations

Libraries, frameworks, and standards provide guidelines and abstractions to help the programmer with the technical details of SM/DM PP. However, the programmer still has to consider performance issues, and some of these issues are transversal to the SM and DM models. This section covers only performance issues of the multithreading and message-passing models. To extract performance from applications, the programmer needs to consider, details such as load balancing, task granularity, and synchronization/communication overhead.

A good load balancing aims for distributing the work among processes/threads in a manner that minimizes their execution time. Unfortunately, sometimes is not enough to just statically divide the total of parallel tasks evenly among processes/threads. Applications may be running in machines with different work rates, to deal with such variability the programmer can use a dynamic work distribution. Libraries such as OpenMP and TBB offer dynamic scheduling mechanisms. However, this is not the case for the MPI standard.

Naturally, a dynamic scheduler has a higher overhead than a static scheduler. Thus, programmers should ensure that the performance gain of using a dynamic scheduler overcomes its additional communication/synchronization overhead. Furthermore, a dynamic scheduler might also be less cache-friendly than a static scheduler, because it promotes irregular data access patterns. In applications running in homogeneous environments, where the parallel work can be quantitatively determined upfront, the use of static distribution might be preferable. However,

with a static approach, the performance might suffer if the application is migrated to different environments. Load balancing is also influenced by factors such as the nature of the workload, the parallel implementation, and the task granularity.

Task granularity can be defined as the time it takes a process/thread to execute a single task. In the literature [ERAEB05, RJ14, MEB99] terms such as *fine*, *medium*, and *coarser* granularity are sometimes used to categorize the task size, along with some formulas to calculate it. The principle is simple; tasks should have enough granularity to overcome the overhead of the parallelism. Moreover, granularity should also promote load balancing. Often choosing the task size is a trade-off between the number of parallel tasks and communication/synchronization overhead. A decrease in the task size leads to an increase in the number of parallel tasks, which makes it easier to achieve a better load balancing. However, an increase in the number of parallel tasks might increase the communication/synchronization overhead, especially with dynamic schedulers. Moreover, an increase of the task size might reduce the communication/synchronization overhead, but it can lead to load imbalance. Usually, parallel tasks in a DM parallelization should have a larger granularity than in the SM parallelization, because of the overhead of communication between processes.

In a SM model, threads share variables that can be concurrently updated; without proper handling, the state of these variables can become inconsistent. Therefore, programmers need to deal with data dependencies inside the parallel regions and ensure the consistency of the application state. To ensure the correctness of parallel applications, OpenMP and languages such as Java, offer synchronization mechanisms (*e.g.*, locks). However, coordinating threads have the disadvantages of locking overhead/contention, restraining some compiler optimizations and reducing the amount of code executed in parallel. If there is too much synchronization between parallel tasks, the application may perform worse than a comparable sequential one [PBB+06]. Nevertheless, sometimes the programmer can use finer-grained synchronization mechanisms (*e.g.*, atomic operations and read/write locks) or even replace the synchronization altogether with private copies of the *problematic* share data.

Performance in DM can be affected by the communication overhead and the dependency on the hardware used to establish this communication. The programmer should aim to reduce the number of messages exchanged and the time spent waiting for them. Whenever possible, the programmer should try to overlap communication with computation, for instance, using asynchronous communication instead of synchronous. However, the use of asynchronous communication makes the parallelization complex and error-prone, and it might not be easy to achieve a good overlap ratio. In some cases, it is faster if the process itself performs the work, rather than wait for the result of another process. Finally, when possible, collective communication routines should be used instead of point-to-point communication, since the former can be specially tuned for communication among several processes.

Chapter 3

Aspect Oriented Programming

3.1 Overview

Software development is known for being a complex and multidisciplinary process, which typically involves the cooperation of people with different backgrounds. With the aim to improve the software development process an extensive range of methodologies, programming languages tools and so on, have been researched and proposed.

Modular programming [Par72a], currently widely used, is one of the most advertised software design approaches. This approach advocates for the division of software into smaller and more manageable modules. Ideally, these modules are independent, interchangeable and cope with a well-defined responsibility (*i.e.*, *single responsibility principle* [Mar03]). Consequently, these individual modules when integrated together fulfill the desired requirements [SWS12].

Most applications are, indeed, composed by a set of modules that work cooperatively to address a variety of requirements and follow some of the modular programming guidelines. However, in complex systems, frequently these guidelines are hardly followed, and consequently, the application modules end up containing mixed responsibilities [GL03].

Modularity in object-oriented programming (OOP) is achieved through the concept of object, which represents an entity with state (*e.g.*, variables), behavior (*e.g.*, methods) and the responsibility to cope with a given concern[1]. However, some concerns in OOP cannot be encapsulated in a specific object; instead, they are spread transversely across distinct objects. Those kinds of concerns are named in the literature as cross-cutting concerns (CCC). Some of the most well-known CCC are:

[1] *"A concern is some functionality or requirement necessary in a system, which has been implemented in a code structure"* [GL03].

Reliability logging [CG02], profiling [Kis02] and checkpointing [MS11, AB08];

Security authentication and authorization [B.04, JBo];

Performance caching [Lad09, Kis02], concurrency [CSM06] and parallelism [MS11];

Correctness business rule implementation [CJ03] and synchronization [HNP98].

In the checkpointing example, the operation of saving the application state in the disk can include the checkpointing of different objects in different execution points. Consequently, some objects are polluted with an additional concern different than the one that they were initially supposed to handle. This mixing of concerns results in scattered and tangled code [GL03], which decreases the software quality, reusability and scalability.

In the context of OOP, techniques such as dynamic proxies and configuration files have been diligently used to solve the CCC issues [Lad03]. Dynamic proxies have the drawbacks of being inflexible [BS03], intrusive, complex [Lad03] and of introducing high overheads [BS03] (typical of a reflection-based techniques). A XML deployment descriptor (*i.e.*, a configuration file) has been used, for instance, in the JavaBeans framework to segregate CCC (*e.g.*, authentication and transaction management) from the business logic. However, such an approach is domain-specific/dependent [Lad03] and, consequently, not portable across different domains and CCC. Furthermore, the XML files have to be painstakingly specified since they are highly error-prone. To make matters worse, if the framework accesses a field that the programmer either mistyped or forgot to write runtime errors may occur [ST15].

In 1997, following the line of reasoning of approaches such as Composition Filters [ABV92], DEMETER (adaptive programming) [LR88] and Subject-Oriented Programming [HO93], Gregor Kiczales and his colleagues proposed the AOP to modularize the CCC [Kic96].

AOP introduces an extra layer of modularity, abstractly a layer of transversal functionality with the creation of aspects. In OOP languages aspects are tantamount to the notion of classes, in the sense that they can be interpreted as modules. Analogously, AOP is a modular approach to CCC as OOP is a modular approach to common concerns. Studies (*e.g.*, [BH08, Ml07]) have shown that AOP benefits software design with:

- an increasing of code legibility;
- the reduction of the amount of code;
- the improvement of application maintenance and evolution;
- a reduction of the cost and software development time;
- the enhancing of code reusability.

The concepts of AOP can be applied to different programming paradigms besides OOP (*e.g.*, functional [DWWW05, SW07]). An AOP system, according to Filman and Friedman [FF00], must have two features, namely *quantification* and *obliviousness*. AOP relies on a mechanism (*i.e.*, pointcut) to attach additional behavior to multiple execution points in a base program (*i.e.*, quantification) in a way that the base program is unaware of that additional behavior (*i.e.*, obliviousness). However, in the past years, with the AOP community growth and research development, some authors [KG02, CRB04, Fab05, RM06] have questioned the restrictiveness of the properties empowered by Filman and Friedman. First, regarding the terminology itself, according to C. Clifton [Cli05] non-invasiveness is being preferably used instead of obliviousness since the former is less restricted than the latter. Both terms imply that AOP can introduce additional behavior to the program without invasively introducing code to that program. However, non-invasiveness does not imply obliviousness by the base program of that additional behavior. In other words, obliviousness implies non-invasiveness but not the other way around. Second, the inelasticity of the properties themselves. For instance, Awais R. *et al.* [RM06] argue that, although desirable, aspects are more about abstraction, modularity, and composability than quantification and obliviousness.

Controversy aside, literature has shown that one can see quantification and obliviousness as desirable features but not as strict rules [RM06]. Some may not want/obtain obliviousness [RC03] or even quantification for that matter [CRB04]. What most people would agree upon with is that abstraction, modularity, and composability, among others, are the holy grail of software design. In fact, such software properties are the motivation behind the work presented in this thesis. Moreover, there are in literature other classifications besides the ones provided by Filman and Friedman [KM99, RM06]. For example, Ramnivas L. [Lad09] presents an informal model with features that an AOP system should have. According to this model an AOP system should match several of the following features: - the ability to identify and select execution points of a given application; - the ability to change the application execution flow and/or its static structure; - and, following the same philosophy of OOP, the ability to encapsulate the CCC into specific modules (referred in AOP by aspects).

There are several proposed AOP extensions for different programming languages, for instance, AspectJ [KHH+01] (Java extension), AspectC++ [SGSP02] (C++ extension), AspectH [Meu97] (Haskell extension). Moreover, AOP concepts have also been implemented with frameworks [PSR05], such as JBossAOP [JBo] and SpringAOP [Spr].

3.2 Tangling and scattering in parallel programming

Tangling occurs when the same block of code (*e.g.*, method and class) is used to implement more than one concern, while scattering occurs when the code to implement one concern is spread across multiple modules. Both symptoms (*i.e.*, tangling and scattering) are related and are a menace to the fundamentals of OOP [GL03]. Furthermore, these symptoms make the code less legible, reduce the software modularity, and consequently, reduce the software quality. AOP aims to decrease the tangling and scattering phenomenon, through the modularization of CCC into modules (coined as *aspects* in AOP terminology).

```
1  ... class MD implements ... {
2    IParticle [] particles = ...
3    int numThreads = ...
4    ThreadPool threadPool = ...
5    ...
6    public void forceCalculation (...){
7      try{
8           final int processID = MPI.COMM_WORLD.getRank();
9           final int numProcess = MPI.COMM_WORLD.getSize();
10          // Adding work to the Pool
11          for(int threadID = 1; threadID < numThreads; threadID++)
12               threadPool.execute(() ->  forceParallel(...));
13
14          forceParallel(...);  // Master Work
15          processDataReduction(...);
16     }
17     catch (MPIException e){...}
18   }
19
20   private void forceParallel(...){
21        int globalID = threadID + processID * numProcess;
22        int totalWorkers = numProcess * numThreads;
23        for(int pA = globalID pA = 0; pA < NParticles; pA += totalWorkers pA++)
24          for(int pB = pA + 1; pB < NParticles; pB++)
25               particles[pA].forcePairParticles(particles[pB], ...);
26
27        threadPool.callBarrier();
28        threadPool.threadDataReduction(...);
29        threadPool.callBarrier();
30   }
31 }
32 ... class Particle implements IParticle{
33   double fx, fy, fz; // forces
34   ThreadLocal <Double> threadLocalFx = ThreadLocal.withInitial(() -> 0.0);
35   ...
36   public void forcePairParticles(IParticle particleB, ...){
37     ...
38     if(distance(this, particleB) < radius){
39       forceABx = ...;
40       fx = fx + forceABx;
41       particesB.fx = particesB.fx - forceABx; // Newton's 3rd Law
42       ...
43       threadLocalFx.set(threadLocalFx.get() + forceABx);
44       particesB.threadLocalFx.set(particesB.threadLocalFx.get() - forceABx);
45       ...
46     }
47   }
48 }
```

Figure 3.1: An example of tangling and scattering problems in parallel programming.

The tangling problem is common in parallel programming models that introduce the PRC invasively in the base code and is even more pronounced when different types of parallelization are combined (*e.g.*, SM + DM). One of the main goals of parallelization is performance, so it is common to duplicate data structures to minimize synchronization overhead, resulting in more complex tangling issues. Furthermore, multiple parts of an application might be parallelized, consequently resulting in scattering problems. To illustrate the tangling and scattering issues, let us use the code of a simplified hybrid parallelization of a molecular dynamic (MD) simulation presented in Figure 3.1. To parallelize the MD simulation intrusive changes to the *MD* and *Particle* classes were made. The lines of code colored with black, orange and red represent the sequential (*i.e.*, base code), SM and DM concerns, respectively. The grey lines are sentences that existed in the sequential version but had to be excluded from the parallel version.

The code of Figure 3.1 calculates the forces between pairs of particles (line 25) within a given radius (line 38). The force calculation between a particle and the remaining is assigned in a round-robin fashion to every thread, across all processes (line 23). Additionally, to avoid data races during the update of forces (lines 40 and 41), each thread has a local copy of the forces (line 34). In the end, within each process, threads synchronize and reduce their local forces (lines 27 to 29) so that, afterward, the master thread of each process can reduce the force local data among processes (line 15).

The intrusive introduction of the PRC (lines of code colored as orange and red) made the base code more complex and harder to understand. It is worth to mention that the code related to the creation, termination, and synchronization of threads/processes was omitted. Thus, the tangling in this example is even more grotesque in practice. To test different tuned optimizations, for instance, to deal with load unbalancing (*e.g.*, dynamic) or data races (*e.g.*, atomics) the code has to be rewritten. Moreover, if these optimizations were applied elsewhere any future modification to their reasoning would provoke code modifications in different locations of the application.

In Figure 3.1 the PRC (*e.g.*, task creation and work assignment) are scattered across different base classes (*MD* and *Particle*). This approach is undesirable in systems with a large number of classes. These problems, among others, expose the inherent lack of modularity of OOP to address PRC. The previously mentioned problems also restrict the use of some of the OO features. For instance, if the class *Particle* or *MD* were to be extended and their methods to calculate the force overridden, the parallelism-related code could be lost. Another problem is the use of Java interfaces, for example, a particle can be represented by an interface with several implementations, and therefore the PRC would have to be injected in all these implementations.

A better solution than intrusively adding the PRC into the domain code is to modularize these concerns by encapsulating them into separate independent modules that can be composed with the domain code. This strategy is explored in this thesis with AspectJ, where the CCC can be modularized using aspects.

3.3 AspectJ

A team of researchers from Xerox corporation led by Gregor Kiczales decided to create a Java extension named AspectJ [KHH+01]. Later on, the AspectJ project was transferred to the open source community at eclipse.org [Lad03] and today is led by Adrian Colyer. The AspectJ 1.0, released in 2002, was a significant combustion source for the explosion of AOP in the 2000s decade, reaching its hype peak around 2004 [Lad09]. In the years that followed, the hype decreased with the natural dissipation of its exoticism, but the language maturation and the community acceptance followed the opposite trend. Many were the factors that contributed to this acceptance growth, namely:

- the IBM investment on the technology [Lad09] and its use on the Websphere framework;
- the introduction of annotations with Java 5, which enabled an easy, clear and transparent approach for the join point selection. An approach used by the project AspectWerkz [Bon04], which was merged with AspectJ (AspectJ 5) [AtJF]. Thus, allowing AspectJ users to also take advantage of annotations [Lad09];
- compilation process optimizations, new weaving models (*e.g.*, load-time weaving) and better language development tools (*e.g.*, IDE integration) [Lad09];
- and the integration of the technology by the Spring framework [Lad09, SK10].

Nowadays, AspectJ is no longer in an embryonic stage, is well established in the community and is regarded by many as the most predominant implementation of the AOP concepts [GL03, Kis02, Lad09]. Hence, it is widely accepted as a *de facto* standard for AOP [SC12]. So, it is not uncommon to see the terms AOP and AspectJ used interchangeably in the literature. Nevertheless, AOP refers to a paradigm whereas AspectJ is an implementation of the concepts underlying that paradigm. One can see the relationship between AOP and AspectJ with the same eyes that one sees the relationship between OOP and Java.

Next, we present some of the basic concepts and features of AspectJ, namely: - its join point model; - *pointcuts*; - *advices*; - inter-type declaration; - and finally *aspects*.

3.3.1 Join point model

The concept of join point refers to an identifiable point in the execution control flow. AspectJ restricts the granularity and the types of join points that can be selected. Limiting the access to more stable constructors allows for better control of the complexity and side effects of using a broader join point model [Lad09]. For simplicity purposes, in the context of this thesis when we refer to join points, we are referring only to those available in the AspectJ's join point model.

It is out of the scope of this work to explain all the peculiarities of the AspectJ design decisions. Nonetheless, we will present some of its features, starting with the most relevant join points[2] for the understanding of this thesis, namely: - method call and execution and field get and set.

AspectJ differentiates between a call and an execution of a method. The difference between these two is directly related to its location. The *call* join point is located in the class that invokes the method whereas the *execution* join point is located in the class where resides the actual method implementation. In the code of Figure 3.1, the *call* and *execution* join points of *forcePairParticles* method are in lines 25 and 36, respectively. This distinction, besides affecting the way the matching of join points is performed, when dealing with inherited and overridden methods may have what some call unintuitive [BTF04], surprising [AHO+07] or strange behavior [Mil04]. These differences are explained in the next section. Nevertheless, such behavior is well defined and is related to AspectJ own designing strategies [AESC08].

The AspectJ join point model includes also reads/writes to fields of objects (*i.e.*, field get and set join points). The instruction *fx = fx + forceABx;* (line 40 of Figure 3.1) contains a set (*fx = ...*) and a get (*... = fx + ...*) join point of the field *fx*. When the fields are objects themselves (*e.g.*, an array) their get/set join points only include the reads/writes of their memory references. For instance, the instruction *particles[pA] = particles[pB];* contains a get but no set join point[3].

With AspectJ, it is not possible to intercept the accessed indexes of an array or the value assigned to them, even though works such as [CC07] proposed strategies to do so. Moreover, AspectJ also does not contemplate join points such as loop *parameters* (*i.e.*, begin, end, and stride), *if/else* conditionals, and primitive local variables. The work in [HG06] proposed an extension to the AspectJ join point model to include loops and conditionals and the work in [JMOA08] to include local variables. The AspectJ join model uses a conservative approach; the previously mentioned join points are not part of that model for reasons such as, not being useful enough, too complex or inflexible to be implemented and too sensitive to insignificant changes in the base code [Ecl18a]. For example, let us imagine that AspectJ can uniquely identify the conditional statements from lines 1 and 2 of Figure 3.2. One developer uses AspectJ to intercept the evaluation of the *if* statement, days later a second developer decides to make the code more readable[4] by changing it to the one in Figure 3.3. Although the codes of Figures 3.2 and 3.3 are semantically the same the result of the join point intercepted by AspectJ is not. Normally, Java programmers do not expect if/else statements to be used as hook points to add functionality.

[2]The complete list of join points can be seen in [Sema].

[3]To be considered a set the instruction would have to change the actual memory reference of the object, for example, *particles = new Particle[N];*.

[4]This is a hypothetical scenario. We do not claim that one style is more readable than the other.

```
1 if(list != null && !list.isEmpty) {...}// action 1
2 else {...}// action 2
```

Figure 3.2: Conditional statements join points before refactoring.

```
1 if(list == null || list.isEmpty) {...}// action 2
2 else {...}// action 1
```

Figure 3.3: Conditional statements join points after refactoring.

3.3.2 Pointcuts

The join point model defines which points in the execution flow can be intercepted but to capture them AspectJ provides a mechanism, named as pointcut. A pointcut captures and exposes (limited) context of the join points. The pointcut is as follows:

$$\texttt{pointcutname}([\texttt{parameters}]) : \texttt{designator}(\texttt{ajoinpoint});^{5}$$

To correctly identify different types of join points, AspectJ provides appropriate designators. From these designators, the most relevant for this thesis are *call/execution (pattern)* and *get/set (pattern)*, where *pattern* is the criteria chosen to select the desirable join points. These patterns can be, among others, signatures of methods (*e.g.*, *call(public int Particle.forcePairParticles(...))*).

AspectJ offers the possibility to select patterns through the use of wildcards such as the character '*'. For example, the signature *call(private int ClassX.*())* will intercept all the private methods, without arguments, that returns an integer and belong to the class *ClassX*. It is also possible to use wildcards to abstract the number of arguments (*e.g.*, ".."), to match subtypes throughout a hierarchy of classes (*e.g.*, '+') and so on [GL03].

Besides the previously mentioned designators, AspectJ also provides additional designators that can be combined with the above ones:

- program text-based [GL03, Semb], also known as lexical-structure based [Lad09] (*e.g.*, *within* and *withincode*);
- control flow-based [GL03, Semb, Lad09] (*e.g.*, *cflow*);
- state-based [GL03, Semb] that are in [Lad09] subdivided into two categories: execution-object (*e.g.*, *this* and *target*) and arguments (*e.g.*, *args*).

[5] Syntax transcript from [GL03].

Logical operators such as && (and), || (or) and ! (not) can be used to combine or negate join points/pointcuts.

Control flow-based designators are the only ones that receive as an argument another pointcut (*e.g.*, *cflow (pointcut)*). Pointcuts that use these designators cannot be fully determined at compile-time, which might result in additional runtime overhead [BHMO04]. Therefore, those concerned with performance should use this pointcut sparingly.

The pointcuts *within(Type)* and *withincode(Method/Constructor)* intercept join points that exist inside a given type and method/constructor, respectively. The pointcut *this(Type)* filters join points according to the object type currently being executed, whereas the *target(Type)* filters according to the type of the target object. Besides filtering the join points to be matched based on some object, these pointcuts also expose that object, which allows the programmer to access the content of that object. Finally, the *args(set of arguments)* pointcut filters join points according to a set of arguments and at the same time exposes these arguments.

In AspectJ terminology, pointcuts that capture join points that can be ascertained at compile-time are named as *statically determinable pointcuts* (*e.g.*, *call/execution* and *set/get*)[GL03, Lad09], consequently pointcuts that rely on dynamic context (*e.g.*, *cflow*, *this/-target* and *args*) are excluded from this category. The list of AspectJ pointcuts and designators [Semb] is fairly extensive to be fully described in this thesis. Notwithstanding, Figure 3.4 illustrates some examples of pointcuts (based on the code of Figure 3.1).

```
1  pointcut getBarrier() : call (public void ThreadPool.callBarrier());
2
3  pointcut getParticleB(IParticle particleB) :
4            execution(public void forcePairParticles(IParticle, ..)
5            && args(particleB, ..);
6
7  pointcut getBarrierInMD() : getBarrier() && within(MD);
8
9  pointcut getFx() : (get(* double Particle.fx) || set(* double Particle.fx))
10                 && withincode(public void IParticle.forcePairParticles(..));
```

Figure 3.4: Examples of Pointcuts.

The first pointcut intercepts the calls to the *callBarrier* method of the class *ThreadPool*, whereas the third pointcut further restricts these interceptions to calls that happen inside the class *MD*. The second pointcut intercepts the execution of the *forcePairParticles* method and exposes its first argument, whereas the fourth pointcut intercepts, within the *forcePairParticles* method, the join points resulting from the reads and writes to the field *fx* of the class *Particle*.

```
1  void setArray(int index, int value) {array[index] = value;}
2
3  int  getArray(int index) {return array[index];}
```

Figure 3.5: Accessing array through methods.

As noted in [HG06] pointcuts to intercept the loop range and the accessed array positions would be useful for parallelism. The programmer is restricted to AspectJ join point granularity and the expressiveness of the pointcut designator. To intercept join points that are not part of the AspectJ join point model either additional extensions have to be used (*e.g.*, LoopsAJ [HG06] for loops and [CC07] for arrays) or code refactors that expose these join points have to be made. The former approach has the disadvantage that those extensions are not officially supported by the language or might not even exist for the desired join points. In most cases, the latter approach only requires method refactoring [cat]. For instance, to intercept the accessed array indexes and their content, the programmer can perform these accesses through method calls (*e.g.*, Figure 3.5). Because the index and the value to be assigned are parameters of a method AspectJ can intercept and expose them.

3.3.2.1 Call *vs.* Execution

There are some differences between the *call* and *execution* pointcuts that are worth to be explained in more detail. Arguably, the most important is how these pointcuts match the join points to be intercepted. If the method declaration, used in these pointcuts, does not contain the object type where the method call/execution occurs (*e.g.*, *execution(public void forcePairParticles(..)*) then these pointcuts intercept the call/execution of the methods that match the method declaration. However, when an object type is also specified (*e.g.*, *call(public void ThreadPool.callBarrier())*), let us labeled that type T, the *call* pointcut only considers the methods in objects with the same static type as T or its subtypes, whereas the *execution* pointcut does the same but using the dynamic type instead. Furthermore, regarding the method signature, the *execution* pointcut only tries to match it with methods of the class T or its subtypes, while the *call* pointcut also matches with those from the super-type of T [AESC08]. More in-depth analysis with illustrative examples can be found in [AESC08]. Nevertheless, defining the type T in a *target* pointcut instead of the method declaration allows, for instance, to use a *call* pointcut that evaluates the dynamic type of T, instead. For example, using *call (public void callBarrier())* && *target(ThreadPool)* instead of *call (public void ThreadPool.callBarrier())*.

Also worth mention that a *call* pointcut does not intercept explicit super calls (*e.g.*, *super.f*). Moreover, while *this* and *target* objects are the same in the join points intercepted by the *execution* pointcut, the same might not be the case with a *call* pointcut. For example, the *this* object of the pointcut *getBarrierInMD* (line 7 of Figure 3.4) is an instance of the class *MD* whereas its *target* object of the class *ThreadPool*. The *withincode* pointcut matches join points that exist inside a method, therefore it does not make sense to combine it with an *execution* pointcut. However, combining *withincode* and *call* pointcuts might be useful, for example, to intercept recursive method calls *call(void fib(int))* && *withincode(void fib(int))*.

3.3.3 Advices

Pointcuts capture join points with the aim of adding actions to them. These actions are applied through the use of AspectJ's mechanism termed *advices*. In literature [Lad09] the resemblance between methods and *advices* is commonly used to ease the grasp of this new concept. Indeed, *advices* share many characteristics with methods, namely signatures (*e.g.*, a pointcut), may or may not have arguments, can declare local variables, return values, throw exceptions, call other methods, and so forth [Lad09]. Nonetheless, *advices*, unlike methods, cannot be called explicitly, do not have access type modifiers, are nameless mechanisms[6], to name a few differences.

In AspectJ, there are three types of *advices*, *before*, *around*, and *after* that can be distinguished by the moment when they enter in action relatively to join points. Ergo, *before*, *around*, and *after* define actions to be executed before, *instead*, and after a specific pointcut is triggered, respectively. The *advice around* is the only one that can return a value and use the *proceed*, which allows the execution of the original join point. The reasoning behind this syntax is that unlike the *before* and the *after* that execute additional actions besides the original join point, the *around advice* will execute a different action instead of the original join point. Hence, the necessity to provide a mechanism that allows the execution of the original join point, if needed. As a matter of fact, with *proceed* the programmer can execute the original join point multiples times and even with different context.

An *around advice* has a returning type that matches the ones from its advised join points. The same applies to the returning type of the *proceed* since it corresponds to the execution of the original join point. Moreover, the *proceed* receives the same number and types of arguments as the *advice* itself. However, it is not mandatory that these arguments have the same values, which gives the programmer the possibility to change some of the advised join point contexts.

Advices can access join points context either by a pointcut that exposes their context and passes it to the *advice* or by AspectJ reflection API. This API provides the *advices* with static and dynamic information stored in special objects (*e.g.*, *thisJoinPoint*). The advantage of this API over directly using pointcuts to expose context is that it provides more information and the possibility of developing more flexible solutions. On the other side, reflection-based techniques are known for their high overhead, and this case is not an exception [Lad09]. However, the runtime overhead is relatively low if the sought context is only of static nature [Kis02]. Nevertheless, if for a given problem the context provided by the pointcuts is enough, reflection should be avoided because, apart from being more costly performance-wise, it is also more verbose and does not provide static type checking [Lad03].

[6]This remark is partially true since it is possible to label an *advices* using annotations (*e.g.*, *@Advice-Name(advice name)*) [Lad09].

Figure 3.6 presents three examples of *advices*; the first two use the pointcuts defined in Figure 3.4, and the last one uses an anonymous pointcut. The first *advice* prints the line in the source code corresponding to the call of the *callBarrier* method of the class *ThreadPool* before that method is executed. The second *advice* intercepts the *forcePairParticles* execution and re-executes it with a different argument value. Finally, the last *advice* logs a message after the execution of the *processDataReduction* method.

```
1  before() : getBarrier(){
2    System.out.println(thisJoinPoint.getSourceLocation());
3  }
4
5  void around(IParticle particleB) : getParticleB(particleB){
6    proceed(new Particle(...));
7  }
8
9  after() :(execution(private void processDataReduction())){
10   log("Reduction among processes finish");
11 }
```

Figure 3.6: Examples of *Advices*.

With AspectJ, not only can the programmer modify the application execution flow but also the application static structure (*e.g.*, class hierarchy). The former type of modification is known as dynamic crosscutting and is performed with pointcuts and *advices*, whereas the latter is known as static crosscutting and is accomplished through the use of inter-type declaration [Lad09, KHH+01].

3.3.4 Static crosscutting

Dynamic crosscutting allows to intercept and add behavior to join points belonging to a specific target type (*e.g.*, class) without polluting the domain code. However, in some situations, to correctly implement the CCC with aspects, the programmer might require state and functionalities that are lacking from the target types. Therefore, and since it is desirable that the domain types do not contemplate state and functionalities related with CCC, AspectJ provides static crosscutting that enables the insertion of new state and functionalities into target types without invasively changing them. Static and dynamic crosscutting work cooperatively to enable the encapsulation of CCC and the development of domain code unaware of aspects' requirements. AspectJ provides static crosscutting features such as inter-type declarations (ITDs) and aspect precedence (detailed in Section 3.3.5).

ITDs can add extra functionality and state to types such as classes, interfaces, and aspects. With ITDs, it is possible to add methods (*e.g.*, concrete and abstract) and fields to all those types, and constructors to both classes and aspects. Furthermore, ITDs also allow type-hierarchy modifications (*e.g.*, *declare parents* : ...), such as declaring that a given type extends another type or implements a particular interface [GL03].

```
1  declare parents  : Particle implements Serializable;
2
3  public int Particle.counter = 0;
4
5  public void Particle.saveDataInDisk(Particle a) { ... }
```

Figure 3.7: The use of inter-type declarations in a checkpointing example.

Let us imagine that the programmer wants to save an object into a disk after a specific method within this object has been executed a certain number of times. Figure 3.7 shows the sketch of a possible solution for this example using ITDs, where *Particle* is the object to be saved. In the lines 1, 3 and 5 AspectJ makes the class *Particle* implement the interface *Serializable*, adds to that class a new instance variable and a method to perform the checkpointing, respectively. This solution removes the checkpointing CCC code from the class *Particle* and also encapsulates into an aspect all the required behavior and state to accurately represent the checkpointing CCC. Furthermore, this aspect can be expanded to checkpoint other classes as well, and thus centralizes the checkpointing CCC into a single location easily modifiable and maintainable.

The members (*e.g.*, fields and methods) introduced with ITDs are treated as if they have been declared directly in the target type [GL03]. However, their access specifiers are tied to the aspect that declared them [Lad09]. The access specifiers of the members can only be either private or public. With the former members can only be accessed by the aspect that declared them, whereas with the latter members can be accessed by other types as well. In the example of Figure 3.7, the *saveDataInDisk* method can, for instance, be explicitly called by other methods of class *Particle*. However, such situation introduces interdependence between domain classes and aspects, which, among others, will inevitably affect modularity.

The functionality and the state added by ITD, for the most part, respect Java own design rules (*e.g.*, overriding methods). The introduction of concrete methods to interfaces was one of the exceptions between what was allowed by Java and what was possible with AspectJ. However, with Java 1.8 that is no longer the case; thus, one can add concrete methods to interfaces also with plain Java.

3.3.5 Aspects

A class is the modularization unit of Java, which, in a broader sense, holds information about the state, behavior and how objects (*i.e.*, instances of that class) are created. Similarly, AspectJ has its modularization unit labeled aspects, designed to encapsulate join points, pointcuts, *advices*, static crosscutting constructors, and so on. Aspects and classes share several characteristics, namely, aspects can: - extend abstract aspects or classes, and implement interfaces; - have variables, methods, and nested aspects (defined as static); use the same access specifiers as classes, and these specifiers follow the same Java rules as well.

Aspects resemble classes in many ways; however, there are differences between the two concepts. Aspects, according to R. Laddad [Lad09], differ from classes mainly on their instantiation model, inheritance restrictions and the possibility of using the *privileged* access specifier. Unlike classes, aspects cannot be explicitly instantiated. Moreover, aspects can extend abstract aspects but not concrete ones, and aspects can extend classes but not the other way around. Aspects by default, as well as classes, cannot directly access private members of another class. However, it is possible to declare an aspect as *privileged* and allow it direct access to classes' private members without using intermediate methods to do so. Nevertheless, *privileged* aspects should only be used when strictly necessary, since it establishes a dependency between aspects and classes (*i.e.*, changes to the classes might forces changes to the aspects). Thus, in literature, some authors recommend for the moderate use of *privileged* aspects [GL03, Lad09, CCHW04] and some regard it as bad style [MF04].

When not declared otherwise, AspectJ creates only one instance *per* aspect, hence, by default aspects are singletons. Although aspects cannot be explicitly instantiated (*e.g.*, with the keyword *new*), it is possible to create multiples instances of the same aspect. These non-default aspect instantiation mechanisms are useful to replicate state and associate it with an event or object. For instance, instead of creating a *hashmap* to associate an object with the state to be replicated, the programmer can rely on AspectJ to do so. Providing mutual exclusion of an object is a classic example [Lad09, CCHW04] of the usefulness of aspect instantiation *per* object. The programmer creates an aspect with a lock and methods to acquire and release that lock, and then associates this aspect with the desirable object to provide mutual exclusion.

An AspectJ application might be composed by *advices* that intend to work with the same join points (*e.g.*, two or more pointcuts that intercept the same method call). These *advices* might be in the same aspect or spread across different aspects. Therefore, foreseeing such scenario, AspectJ has rules to deal with intra- and inter-aspect *advice* precedence. The precedence of conflicting *advices* in the same aspect is ascertain based on their type and lexical order in the aspect [Lad09], namely:

- *"If either are after advice, then the one that appears later in the aspect has precedence over the one that appears earlier"* [pre];
- *Otherwise, then the one that appears earlier in the aspect has precedence over the one that appears later.* [pre].

Naturally, all the *before* and *after* advices will be executed before and after the join points that they advice regardless of their order in the aspect, respectively [Lad09]. However, if there is an *advice around* that does not call *proceed* from all the *advices* to be executed next, only those with a higher precedence than that *advice around* will be executed.

When *advices* matching the same join points belong to different aspects, their precedence will depend on the precedence of their aspects. For aspects precedence the only rule is that derived aspects have a higher precedence than their base aspect, allowing these derived aspects to override the behavior of their base aspect [GL03]. For the remaining situations, the precedence is unpredictable [Lad09, GL03]. Nevertheless, the programmer can deal with undefined precedences by declaring himself the desirable precedence (*e.g.*, *declare precedence : A, B, ...*). Declaring that an aspect A has precedence over an aspect B means that, when advising the same join points, all the *advices* from A have precedence over those from B. This implies that: - all the *before advices* from A are executed before those from B; - all the *around* advices from A encloses those from B; - and all the *after advices* from A are executed after those from B.

The lexical order of the *advices* within an aspect might conflict with the precedence rules of AspectJ causing a *"circular advice precedence"* error [LHBL06]. Similar issues can also happen with the precedence among aspects. Nevertheless, the AspectJ notifies the programmer of such problems at compile-time and some IDE (*e.g.*, Eclipse) even before compilation. Although the intra-aspect precedence rules look complex at first, in practice they are not. In our experience, coding the aspects to have their *advices before* in the beginning, followed by the *arounds* and the *afters* helps to reason about the order in which the *advices* will be applied. With that approach, the programmer can look at the lexical order of the *advices* in the aspect to understand their execution order. Such an approach is especially useful when building a library of aspects because it might not be possible to foresee which join points will be advised in the future.

From the inter-aspect precedence rules, we can anticipate that aspects with many *advices* might become a problem. The programmer may want that only a subset of *advices* from one aspect takes precedence over the *advices* from another aspect. Thus, in some cases, it is preferable to create more and smaller aspects with fewer *advices* to easily manage possible precedences.

3.3.6 Weaving

The AOP concepts were designed to be employed as a complement to a base language, providing mechanisms to encapsulate CCC into independent modules. The concepts are provided as extensions (*e.g.*, AspectJ) of a base language (*e.g.*, Java), allowing the independent development of domain code and aspects. These extensions inject the code from the aspects into the desirable join points in the domain code [HH04].

In AspectJ, aspects are merged with Java classes through a process designated as *weaving*. This process will produce bytecode as a result of applying the aspect code into the join points of the classes that it affects [Lad09]. The AspectJ compiler (*e.g.*, ajc) receives classes and aspects in either source code or bytecode format. The *advices* and the pointcuts filled with the join points to be intercepted indicate to the *weaver what* crosscutting behavior to add and *where* to add it, respectively. The most commonly weaving approaches used with AspectJ are:

- compile-time weaving [GL03, PSR05, Wea] also known as build-time source code weaving [Lad09];
- post-compile weaving [Wea] also known as link-time weaving [GL03] and build-time binary weaving [Lad09];
- load-time weaving [GL03, Wea, Lad09];

Compile-time and post-compile weaving combine aspects, both in source form or bytecode, with Java classes before execution. However, the former approach uses the source code of the classes whereas the latter uses the bytecode. Furthermore, the former approach is simpler [GL03, BH02] but requires the presence of the application source code. The load-time weaving occurs when Java's *classloader* is loading the classes. With the load-time technique, the weaving process is delayed until it is strictly necessary (unlike the first two approaches). With AspectJ, it is possible to combine multiple weaving approaches (*e.g.*, weaving some aspects at compile-time and others at load-time).

AspectJ weaving process allows the application of the AOP concepts on top of Java without the need to design an entirely new language from scratch. This fact, along with the variety of weaving models, makes it easier for the Java developers to adopt the AOP.

3.4 Aspects reusability

The concept of reusing knowledge is transversal and of utmost importance to various fields, including software development. During the software development process, one might reuse and combine different trustworthy software instead of creating solutions from scratch [Kru92]. Software reusability can occur in many forms, namely functions, strategies (*e.g.*, design patterns [GHJV95]), libraries [CA08], and so on.

Reusing software improves the reliability of the application (the programmer can reuse code that is stable and proven to be correct), accelerates the development process, reduces software related costs, among others. However, the development of reusable software is harder than non-reusable, since the programmer has to deal with an additional set of requirements [Som10]. For instance, the code has to be abstract enough to work on different projects, preferably without introducing too much complexity, and at the same time having an acceptable efficiency.

AspectJ allows the modularization of CCC into aspect modules; therefore, it is paramount that such knowledge can be reused in different applications, typically through the use of libraries [CSM06] and frameworks [JBo, Spr]. However, AspectJ also introduces its own set of challenges to software reusability [HU01, Wam06, CSM06, KAB07, Byn11], because of the intrinsic nature of its compositional mechanisms. The developer of reusable aspects has to reason about the expected interfaces (*i.e.*, the join points to be intercepted), the grouping and coordination of *advices*/aspects, the parts of the code that should (or not) be extensible, and how to manage the state related with the CCC. Naturally, developing libraries capable of providing SM parallelism adds additional challenges. The developer has to understand, among others, how to handle the internal state of the aspects in a multithreaded environment, and possibly add synchronization to ensure the correctness [CSM06]. Moreover, because the aim of parallelism is performance, the developer must also reason about the trade-off between flexibility and performance of some AspectJ features (*e.g.*, *cflow*).

Typically, the development and use of reusable aspects have to take into consideration three phases, namely the development of reusable aspects itself, the preparation of the target application and the connection between the reusable aspects and the target application. According to Maarten *et al.* [Byn11], there are three main properties that define the quality of the reusable aspects, namely stability, versatility, and easy configuration. Thus, reusable aspects should be: - independent from the target application as much as possible, so that aspects and base code can evolve independently without compromising each other (stability); - abstract enough to be reusable across different applications (versatility); - easy to be adapted to the base code, even by non-AspectJ experts (easy configuration). The developer should identify the join point interface on which the reusable aspects will rely. Ideally, that interface should be abstract and stable enough to provide greater reusability and stability [Wam06]. Preferably, no change in the target

application should be made, however, due to the granularity of the AspectJ join point model, sometimes modifications to the base code are needed to expose the desirable join points (*e.g.*, transform specific statements into methods).

There is in the literature work dedicated to identifying strategies to rationalize and systematize the use and development of reusable aspects [Byn11, HS03, GK01, LJ06, NSPB07] (*e.g.*, idioms). Common idioms include: - abstract pointcuts [HU01, HS03]; - marker interfaces [HC02, HS03]; - annotations [Wam06, Lad09, CSM06]; - among others [HE08, Lad09]. These idioms can be used individually, or combined, to abstract reusable aspects from the application's join points. Maarten [Byn11] provides a comprehensive system of patterns to build stable, versatile and easy to configure reusable aspect libraries. The author explains and suggests which and when different idioms should be used and or combined.

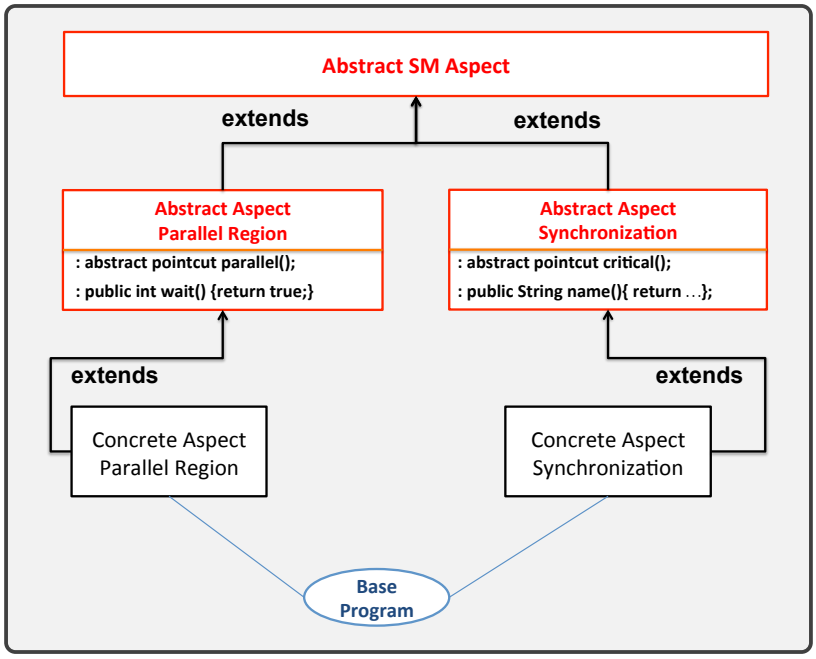


Figure 3.8: An overview of an example with abstract aspects and pointcuts.

With AspectJ the programmer can use and combine mechanisms such as inheritance, abstract aspects/pointcuts, generics, marker interfaces and so on to develop reusable aspects. For example, as shown in Figure 3.8, the programmer can develop abstract aspects (*e.g.*, *Parallel Region*) with abstract pointcuts (*e.g.*, *parallel*) to encapsulate and share behavior and state transversal to their sub-aspects [HU01, HS03]. Later on, for each application, these abstract aspects are extended by concrete ones that encapsulate state and behavior specific to the target application. The mapping between the abstract pointcuts and the join points to be intercepted (*e.g.*, *pointcut parallel() : call (public void forceCalculation(..))* is defined in the concrete aspect (*e.g.*, *Parallel Region*). The concrete aspect works as a bridge between the core of the library of aspects and the target application [GHJV95, Wam06].

The upper levels of the aspect hierarchy (*e.g.*, Abstract SM Aspect) can also share pointcuts with their sub-aspects (*e.g.*, parallel region and synchronization). Hence, the sub-aspects can build pointcuts upon definitions created in the upper levels. A pattern, known as elementary pointcut [LJ06, BLJT07], can be applied to reduce pointcut complexity and increase their reusability, consisting of defining a given pointcut as the aggregation of simpler ones that can be defined in the sub-aspects. Another common strategy to increase aspect reusability is to apply the *template method* pattern [GHJV95] into the *advices'* code [HS03]. With this pattern, *advices* are composed of methods that can be overridden to satisfy different conditions. In Figure 3.8, by default, threads wait at the end of the parallel region, however that behavior can be changed by overriding the *wait* method in the concrete aspect.

Marker interfaces [GHJV95] can be used to signal that a class has join points of interest. Marker interfaces can be useful when the join points to be intercepted can be selected based on the datatype alone. Let us assume that the *Synchronization* aspect of Figure 3.8 synchronizes all the public method calls of a certain class to make it thread-safe. To be reusable that mechanism can use a marker interface (*e.g.*, *IThreadSafe*) that signals the classes with methods to be synchronized (Figure 3.9). Hence, that mechanism can be used in any class as long as the class implements the marker interface. The specification of the classes that will implement the marker interface can be done non-invasively in the concrete aspects using ITD (Figure 3.10).

```
1 abstract aspect Synchronization {
2   pointcut synchronizeMethods() : call (public * IThreadSafe.*(..));
3   ...
4 }
```

Figure 3.9: Marker interface example (abstract aspect).

```
1 aspect SynchronizationConcrete extends Synchronization {
2   declare parent : Particle implements IThreadSafe;
3 }
```

Figure 3.10: Marker interface example (concrete aspect).

Marker interfaces can also be useful when the join points to be intercepted by ITD or pointcuts are from aspects that are not connected (*i.e.*, cannot share the same pointcut or ITD definition). To exemplify those scenarios, let us use the example of Figure 3.7. As we can see that code is not reusable since it is tightly coupled to a concrete class from the domain (*Particle*). However, abstracting that concrete class with a marker interface[7] (*e.g.*, *ICheckpoint*) would make the code reusable. Checkpoint could be provided to different classes by making them implement the appropriate marker interface instead of duplicating the code of Figure 3.7. Let us assume that for fear of deadlocks[8] checkpointing is performed before the synchronization of methods by the *Synchronization* aspect. Without marker interfaces, the appropriate pointcuts

[7]The classes implementing that interface will be the target of the inter-type declaration. Hence, will have the variable *counter* and the *saveDataInDisk* method.

[8]This scenario is unrealistic it is just to illustrate the point.

from the *Synchronization* and checkpointing aspects would be filled with the same method calls. However, with marker interfaces, the checkpoint aspect can internally make all the classes that implement *IThreadSafe* also implement *ICheckpoint*.

```
1 abstract aspect Synchronization <T> {
2   pointcut synchronizeMethods() : call (public * T.*(..));
3 }
```

```
1 aspect SynchronizationConcrete extends Synchronization <Particle>{...}
```

Figure 3.11: Example of using generics (abstract and concrete aspects).

AspectJ 5 introduced generics and annotations that, as marker interfaces, can be applied to join points that can be selected based on their target type. The class thread-safe example could have also been implemented with generics (Figure 3.11) or annotations (Figure 3.12).

```
1 @Retention(...) public @interface IThreadSafe {}
2
3 abstract aspect Synchronization {
4   pointcut synchronizeMethods() : call (public * (@IThreadSafe *).*(..));
5 }
```

```
1 aspect SynchronizationConcrete extends Synchronization {
2   declare @type: Particle :  @IThreadSafe;
3 }
```

Figure 3.12: Example of using annotations (abstract and concrete aspects).

There are scenarios where annotations or generics can be preferably used instead of marker interfaces [Byn11]. Annotations can be applied to different elements besides only types (*e.g.*, methods and fields) and allow to conveniently pass metadata (*e.g.*, *@parallel(nowait=false)*). In cases where a pointcut needs to match multiple types (*e.g.*, *call (public * T.*(..)) && within(Y)*) instead of marker interfaces generics can be used to more accurately represent the relationship among these types. In such cases, the problem of using marker interfaces (and annotations as well) is that they are defined individually and alone they are meaningless, whereas with the generic aspect the relationship among types is explicitly expressed (*e.g.*, ... *aspect callofTinY* $<$ *T, Y* $>$). Also, the approach with generics would be less error-prone since all the participating types have to be explicitly defined, which is not the case with marker interfaces. There are also situations where annotations or generics cannot be used, for instance, AspectJ does not allow annotations or generics to be used as targets of ITD [Ecl18b] as the case of the checkpoint example (lines 3 and 5 of Figure 3.7). Nevertheless, that code could still be improved by combining generics with either annotations or marker interfaces. In an improved version, as shown in Figure 3.13 that combines generic and marker interfaces, the user should not have to care about which interfaces to implement, all of these details are handled internally by the aspect. Regarding annotations, a more in-depth explanation of their advantages and disadvantages will be provided later in the context of programmability and maintainability.

```
1 abstract checkpointing <T>
2 {
3   interface ICheckpoint{} // Marker interface
4
5   declare parents : T implements ICheckpoint, Serializable ;
6
7   public int ICheckpoint.counter = 0;
8
9   public void ICheckpoint.saveDataInDisk(Particle a) { ... }
10 }
```

Figure 3.13: Improved checkpointing example.

Reusable aspects not only have to define the appropriate abstractions to expose the join point but (might) also need to manage state and behavior (*e.g.*, methods) that is tightly connected with these join points. For instance, in the checkpoint example, the variable *counter* and the *saveDataInDisk* method are, respectively, state and behavior that is tightly connected to the classes that implement the *ICheckpoint* interface.

By default, aspects are singletons and cannot be explicitly instantiated. Nonetheless, it is possible to create (implicitly) instances of the same aspect. AspectJ offers mechanisms (*i.e.*, aspect instantiation) to map the creation of aspects instances to objects, types, and control flows (*e.g.*, a certain method execution). With AspectJ, it is possible to add state to objects through both ITD and aspect instantiation *per* object. However, the former provides state to all instances of a given object, whereas in the latter it is possible to narrow the subset of affected objects to only these that match the join points intercepted by the pointcut used as a parameter of the aspect instantiation mechanism. R. Laddad stated that aspect instantiation *per* object may be preferable to *introduction* when developing reusable aspects (arguing that the former might provide a more elegant solution) and that the choice between the two approaches is a *"balance between elegance and simplicity"* [Lad09].

In some cases, it is desirable to restrict the state and behavior to be added to only a subset of objects or during a specific event. In such cases, some authors considered the use of ITD as being a code smell (labeled *Aspect Laziness*), since its effects are felt in all instances of a given object and last for their entire lifespan [MF05]. Some authors also suggest replacing ITD with aspects that map the state and behavior to the proper join points, a detail description of this refactoring can be found in [Mon04]. Although the aspect instantiation mechanism offers (to a certain extent) a solution for the mapping between state/behavior and join points, we favor the use of internal structures in the reusable aspects to do so. The latter offers to the programmer of reusable aspects more control of the structure used to perform the mapping. In some scenarios, a simple *hashmap* will be sufficient while on others synchronized structured might be needed (*e.g.*, concurrent hashmap)[9].

[9]Until 2013 the aspect instantiation mechanism was not thread-safe [Bug].

Chapter 4

Related Work

4.1 Libraries, frameworks and languages

Multicore systems are widely available in desktop machines, and it is expected that the number of cores in these systems continues to grow over the next decades. To answer to the ever increasing of parallelism capabilities provided by the hardware manufacturers, popular languages such as C/C++ and Java were extended to support that parallelism. Some of the extensions are low-level and provide only a minimal set of constructors to exploit parallelism (*e.g.*, POXIS Threads and Java Thread API). With low-level extensions, the programmer can develop applications with higher performance but at the cost of more programming effort and a greater level of expertise in both the domain- and parallelism- related concerns. Typically, these highly tuned codes are synonymous with high complexity and low software legibility, reusability, and often maintainability. Nevertheless, these trade-offs are acceptable in highly tuned domain-specific libraries (*e.g.*, BLAS [BPP+02]), since the user can transparently benefit from the performance of these optimized implementations. Unfortunately, in many domains, it is not possible to pack these optimized implementations into libraries and programmers have to develop their (own) implementations.

Unsurprisingly, the increase in popularity of PP, and consequent use by non-experts, increased also the demand for higher-level abstractions that can be used to parallelize code of different domains more easily. Libraries, frameworks, and even programming languages can provide these higher-level abstractions (*e.g.*, OpenMP and Intel Threading Building blocks (TBB)). This trend of moving from lower to higher-level abstractions was also felt in the evolution of programming languages themselves to meet the increase of machine power, and consequently software complexity.

4.1.1 SM and annotation-based approaches

TBB is a C++ template-based library that provides abstractions for SM parallelism without the need for special compilers. Some of TBB strongest points are its sophisticated work-stealing scheduler, cache-aware memory allocator, concurrent collections (*e.g.*, hashmap and vectors), parallel patterns [MRR12] (*e.g.*, sort, reduce and *for*), and the flow graph interface. However, TBB does not support alternative scheduling strategies [KSG09], as OpenMP does[1]. Furthermore, the TBB scheduler is suitable for applications with load balancing issues [Rei07], however, in applications without these issues, it might be slower than a static scheduler [CM08].

In TBB to parallelize *for* loops, these have to be turned into method calls (*e.g.*, *parallel_for*) that receive the loop range (*i.e.*, begin, end and stride) and body as parameters. Before the introduction of lambdas in C++, this loop method transformation was considered to be a great hindrance to learn TBB[2] because of the warping of the loop body as a functor [Rei]. Since then, lambdas have reduced the amount of logic needed to define the functor making it more legible.

TBB hides from the programmer some of the lower-level PRC (*e.g.*, explicit thread management and work distribution), increases the level of abstraction by relying on C++ templates and promotes a more structured programming style. However, this requires more program design, which can be cumbersome in legacy code [KSG09]. In TBB, as in traditional PP approaches, the PRC are mixed up with DRC affecting code maintainability and evolution. Hence, making it harder for the programmer to reason about or debug the (sequential) domain code in isolation. Finally, to run the code sequentially, the programmer has to explicitly disable the parallelism through a method call to the TBB library, unlike OpenMP where parallelism can be disabled by simply omitting a compilation flag.

OpenMP hides several lower-level PRC and provides a certain degree of customization of these concerns both through parameterizable annotations. Compared with other intrusive approaches (*e.g.*, TBB), annotations allow for a better separation between domain and PRC, reduce the code complexity and intrusiveness of the PRC. OpenMP promotes an incremental development approach, where PRC are inserted into the source code as annotations without changing the domain code [VGS13]. This last advantage, in particular, makes annotation-based approaches especially suitable for legacy code. However, the main drawback of annotations is that they are restricted to a set of constructors and most of their parameters have to be known at compile-time. Moreover, sophisticated parallelism-related strategies force the use of explicit constructors, such as threads ids, object locks and so on. Consequently, the benefits of annotations are lost in those sophisticated strategies, and the PRC are explicitly mixed with DRC. Finally, OpenMP, as well as TBB, are limited to SM parallelism which limits their scalability.

[1] A comparison between the OpenMP and TBB constructors can be found in [V.11].

[2] James R. said in 2009 that *"It was the toughest thing to teach about using Intel TBB"* [Rei].

The OpenMP standard does not officially support Java, but there are in the literature examples of Java implementations, namely JOMP [BK00], Pyjama [VGS13] and JaMP [KBVP07] (or cluster ClusterJaMP [VBP11]). The last two proposals extend their implementations to contemplate, among others, GUI applications and cluster environments, respectively. In these proposals, the parallelism is expressed through special directives that mimic as much as possible the OpenMP pragmas. These directives are special Java comments that are interpreted by appropriate intermediate software (*e.g.*, precompiler) that injects the desirable parallelism-related constructors. These constructors are injected into the base code in JOMP and Pyjama by their source-to-source precompilers that transform the code accordingly (including calls to their respective runtime libraries) and in JaMP by using the Jackal framework [VHBB01].

Although, JOMP, Pyjama, and JaMP implement, to some degree, the OpenMP standard (varying from version 2.0 to 3.0) they lack some relevant features for parallelism in Java [KVBP08]. In that list, one can include, among others, the extension of the OpenMP data directives to objects and their fields, user-defined reductions, parallelization of loops over Java collections (*for each*) and task parallelism. Moreover, as with OpenMP and TBB, JOMP and Pyjama are limited to SM parallelism. JaMP, however, offers DSM parallelism since it is built on top of the Jackal framework. JaMP offers the OpenMP constructors and Jackal ensures that, through their own modified Java compiler [MMPS09, VHBB01], the resulting multithread applications run in a cluster [VHBB01]. However, unlike JOMP and Pyjama that are pure Java, JaMP lacks portability because of relying upon the Jackal framework [TRE+13]. These OpenMP proposals for Java suffer from the same limitations presented earlier for OpenMP.

Java annotations mechanism introduced in Java 1.5 offers a more robust alternative to comment-like directives. Annotations are an inbuilt Java feature and offer, among other advantages, a more reliable support, do not require extra precompilers to interpret them and can be easily extended by the users. The downside of annotations compared with directives is that their use is limited to well-known code structures (*e.g.*, classes and methods). Hence, implementing OpenMP constructors based on Java annotations, typically, would impose some code restructuring (*e.g.*, method refactoring). Works such as PAL [DPV+08] and Jconqurr [GSBW09] use Java annotations to express parallelism. The former focus on asynchronous method[3] parallelism, whereas the latter combines both annotations and directives to provide parallelism. In Jconqurr the directives are static method calls that, among others, are used to signal meaningful information. For instance, to parallelize a loop *for*, the method enclosing the loop is annotated (*e.g.*, *@ParallelFor*) and the loop is *labeled* by having a directive call right before it (*e.g.*, *Directives.forLoop();*). The directive is used to distinguish the loops that should (or not) be parallelized.

[3] In PAL those methods are not allowed to change object state [DPV+08].

4.1.2 DM

Arguably, the main advantage of MPI over OpenMP is that it can run in both SM and DM architectures[4]. However, MPI typically requires more programming effort and changes to the base code than OpenMP. Most work to implement the MPI standard in Java falls into two categories [TRE+13, TTD03]: - pure Java (*e.g.*, MPJ Express [SCB09] and F-MPJ [TTnD12]); - using an intermediate layer that performs JNI calls to a native MPI implementation (*e.g.*, mpiJava [BCF+99] and OpenMPI [VGRGS13]). The former has the benefit of portability but at the potential cost of performance [TRE+13] and requires more work to adapt these implementations to upcoming features of the MPI standard. The latter is less portable but can easily evolve along with the native implementation and can, potentially, have a performance close to the native MPI implementation. There is also work that implements their own message-passing constructors (*e.g.*, PCJ [NB12]). However, using an established and well-known standard such as MPI offers more reliability and an increase in user acceptability.

OpenMPI, besides being one of the most popular implementations of MPI [VGRS16], also supports Java [VGRGS13] using a JNI-based approach. The Java implementation provides most of the features from MPI 3.1 standard, however, in our opinion, it could be extended to contemplate additional higher-level features that are very handy for programming MPI in Java. Namely the ability to send/receive objects between processes, scatter and gather of some Java collections, among others.

Java, unlike C/C++, does not allow to explicitly manipulate pointers, which makes the building of the messages to be exchanged among processes more cumbersome and complex. In some cases, the programmer has to resort to workarounds (*e.g.*, wrapping primitive data types into an array), or use specific routines (*e.g.*, *slice*), to correctly wrap the message. In MPI applications, it is common to express parallelism by scattering/gathering data across processors. In this pattern, normally the gather and scatter functions are isomorphic (*scatter.gather* = *gather.scatter*). Therefore, instead of manually, and separately, defining the scattering and gathering of the data, the programmer could merely provide the desired distribution type, and the rest would be handled transparently. Especially, since some of these distributions are very verbose and complex (*e.g.*, distribution of matrices by columns) and because frequently the programmer only wants to split/collect the data evenly among/from processes. All of these small, cumbersome, and error-prone technicalities can be hidden behind a high-level API that exploits Java method overload capabilities to extend the functionality while keeping some of the names from the MPI standard routines. Providing the previously mentioned abstractions on top of OpenMPI improves the programmability, reduces bugs and retains the high performance of the low-level OpenMPI implementation.

[4]There are works (*e.g.*, [BME07]) that tried to extend OpenMP to run in DM architectures as well, but officially the OpenMP standard does not support these architectures.

Regarding the tangling and scattering problems, these tend to be more accentuated in MPI than in OpenMP applications. While in OpenMP the parallelism is commonly localized in some specific spots in the domain code, in MPI the parallelism tends to cut across several classes. This issue mainly happens because OpenMP is frequently used for loop-level and task parallelism, whereas MPI for explicit data decomposition. Hence, typically MPI parallelizations require more code design and restructuring.

4.1.3 PGAS and Hybrids (SM + DM)

The Partitioned global address space model (PGAS) implements a DSM model, which aims to combine the scalability of the DM model with the programmability of the SM model [YBC+07]. Usually, the PGAS model implements a SM abstraction over message-passing systems, creating an illusion of global shared memory space. Typically, in the implementations of this model, there is a middle layer that transparently performs the communication among processes. This model is sometimes used as an alternative to hybrid implementations such as MPI with OpenMP.

Titanium [YSP+98], UPC [EGCSY03], Chapel [CCZ07], and X10 [CGS+05] are implementations of the PGAS model. Titanium is a Java-based language, UPC is a C extension, Chapel and X10 (Java-derived) are new PP languages. All of them aim for high performance and productivity, and (excluding UPC) follow the OO paradigm. From the programmability point of view, one of the main advantages of these technologies is their unified syntax to represent SM and DM concerns. Furthermore, the programmer can seamlessly access memory from other processes without having to perform any explicit communication. However, since some of the communications are performed implicitly, extra care has to be taken to not degrade performance as a result of too many remote memory accesses. Hence, even without explicitly performing the communications the programmer might have to reason about them. Even though some of the communications are performed underneath the system, the programmer sometimes (still) has to explicitly perform some form of it (*e.g.*, UPC *upc_all_reduce*).

Languages such as Chapel and X10 provide high-level constructors and follow the tendency of more modern languages, which have compact syntaxes that help to write concise code. Some of these high-level constructors include parallel loops embedded in the language (*e.g.*, *coforall*) and global synchronization among all threads from all processes. For instance, the global synchronization offers a cleaner solution than performing the synchronization explicitly in a hybrid MPI with OpenMP implementation. Hiding parallelism behind the language loops attenuates, to a certain extent, the tangling and scattering problems. Even though such languages hide several low-level parallelism-related details, the programmer still has to invoke some form of thread/process[5] (*e.g.*, *Locales[...]* in Chapel and *Places.places()* in x10). Furthermore, locality

[5] In some of those languages, the concept of thread/processor was replaced by higher abstracts (*e.g.*, tasks and places).

is important for performance, particularly in the PGAS model and is up to the programmer to handle it explicitly. For instance, UPC offers an extended (parallel) *for* that besides the typical parameters also receives the affinity (*e.g.*, *upc_forall(i=0; i <N; i++; i)*). Moreover, PGAS models, as with SM, can have race conditions; therefore explicit synchronization might have to be used. In the end, the PRC are still tightly mixed with the DRC. Furthermore, even though PGAS offers a global view of memory across different processes, SM and DM concerns are still tangled with each other. For instance, in Chapel one have to explicitly differentiate between parallelization inside a target architecture (*e.g.*, *coforall*) and across different targets (e.g., *Locales[...]*).

Despite the developments made in PGAS languages several works still rely on hybrid approaches to combine the simplicity and efficiency of multithreading with the scalability of message-passing. The Parallel Java (PJ) library is an example of such work. PJ relies only on Java to provide features inspired by the OpenMP and MPI standards [Kam07]. The main advantages of PJ are: - having a unified API to express SM and DM PRC; - promotes OO programming style, where some of the parallel constructors are expressed by creating methods and objects (*e.g.*, *execute (0, n - 1, new IntegerForLoop()*); - offers additional useful features, such as parallelization of some *for each* loops, user-defined reductions, exchange of objects between processes; - and being pure Java. The last advantage makes PJ portable and facilitates extending the library by users. However, since PJ is pure Java its message-passing implementation can, potentially, be slower than a Java MPI implementations that use a JNI approach.

One of the disadvantages of PJ is that it requires significant code refactoring. Commonly, in PJ to use a parallel loop, it is necessary to create three objects (*new ParallelTeam()*, *new ParallelRegion()* and *new IntegerForLoop()*), override two methods (the *runs* from the parallel region and parallel *for*) and call two methods (one with the parallel region as a parameter and the other with the body of the loop to parallelize). This refactoring resembles the creation of a typical parallel region in Java using *executors* and *runnables*. Although that last remark can be interpreted as an advantage, since the programming style is natural to Java programmers, in the end, several intrusive lines of code have to be introduced. On the positive side, PJ abstracts from some of the low-level details from the PRC. Notwithstanding, the programmer still explicitly creates a team of threads and assigns them work. Although, PJ facilitates the use of debugging tools (since it is pure Java), debugging and reasoning about the domain code in isolation is simply not possible. Finally, although PJ is inspired in some of the MPI features, it uses its own API making its adoption difficult [TRE+13].

4.2 OO mechanisms and skeletons

Traditional approaches suffer from tangling and scattering problems since the PRC are explicitly mixed with DRC. These issues are especially problematic in applications written in OO languages since parallelism can crosscut multiple classes. These problems impose strong limitations on independent development, long-term maintenance, ability to debug and reason about the DRC in isolation, among others.

A natural choice to implement PRC in OO systems is to use pure OO mechanisms, namely encapsulating these features using inheritance (*e.g.*, *white-box* frameworks as ParadisEO [CMT04]) or decomposition (*e.g.*, *black-box*). Both have their pros and cons, but most importantly, both require changes to the base code and still mix DRC with PRC. Two alternatives to introducing the PRC directly in the domain code is to use inheritance (*i.e.*, subclassing) or composition through the use of the *decorator pattern*. Both techniques can be used to reuse and extend functionality. For instance, the PRC could be integrated with the domain classes by extending or composing these classes with modules providing different parallelism models (*e.g.*, multithreading and message-passing). Some of the disadvantages of the subclassing approach in Java are: - inflexibility because of the impossibility of extending multiples classes; - not scaling when it is necessary to compose multiple features (*e.g.*, hybrid parallelization); - and having to statically modify the classes hierarchy with features that might not be semantically related with the domain code. The *decorator pattern* provides a more flexible and scalable solution than subclassing. However, this approach can lead to a considerable amount of boilerplate code (*e.g.*, forwarding methods and smaller objects). Moreover, in some cases it is hard to implement without major refactoring (*e.g.*, code relying on concrete types), making it hard to debug and reason about the code (*e.g.*, multiple decorators combined in a chain). Although, these approaches reduce the number of PRC mixed with the domain code they still introduce some traces of it. These approaches are presented in more detail in the next chapters.

Lithium [ADT03] and JaSkel [FSP06] are Java frameworks based on the concept of Skeletons [Col91, GVL10], which encapsulate the details of certain parallelism patterns and provide compositional properties. These frameworks require the creation of classes to represent the tasks to be performed and the instantiation of a particular skeleton to coordinate the execution of these tasks. This approach has three main limitations: - the base code is polluted with scaffolding code to redirect the execution to the skeleton framework; - skeletons only encapsulate simple parallelism models (*e.g.*, farm); - it would be difficult to implement multiple parallelism alternatives on top of this approach, as execution issues are delegated to the skeleton framework. YaSkel [NS09], the successor of JaSkel, ameliorates the last limitation by using dependency injection [CI05] to delay the provision of the concrete skeleton implementation until load-time.

4.3 AOP

AOP and FOP [Pre97, BSR04] are programming paradigms based on rewriting mechanisms that can be used to modularize the CCC. AOP provides effective mechanisms to encapsulate individual features, and class refinement[6] [NC08] is a simple but powerful enough mechanism. The main advantage of class refinement is that it is based on well-known OO concepts, which offers a smoother transition and a shorter learning curve. Some argue that those paradigms (*i.e.*, AOP and FOP) can be combined to complement each other [Ape07]. We do not disagree with that point of view; however, we believe that to build a framework for parallelism some of the AOP weaknesses can be attenuated with proper structure and that most of these weaknesses do not affect so much the framework user but more its developer. Moreover, we also believe that some of these weaknesses are not so pronounced in the context of parallelism.

Among others, our framework should be easy to use and maintain, while at the same time flexible. We believe that using a single technique (*e.g.*, AOP) with stable tools and a big community behind it helps to increase the chances of achieving that goal. Hence, we favor the use of AOP, in this context, but acknowledging that using FOP or a combination of the two could also be a suitable solution but with its own set of challenges.

4.3.1 AspectJ extensions

[MK03, AE06, Har06, CC07, JMOA08, XHG09, ACN09, CV13, DM14] proposes extensions to the AspectJ join point model some of which could impact the implementation of parallelism-related modules.

[Har06] proposed a join point for loops, and suggested an *ad hoc* pointcut to intercept array accesses and indices. That work also used tailor-made aspects for the parallelization of some case studies. [DM14] proposes a pointcut for loops that detect if some of these loops are parallelizable, however, because of the intrinsic complexity of automatically detecting parallelizable loops the applicability of that pointcut is very limited. A pointcut for loops looks, at first glance, promising but there are several technical problems associated with it. For instance, the identification of different loops within a method, dealing with loops with ambiguous returning types, fragility (*i.e.*, minor changes to the base code can affect a loop pointcut), and performance (*i.e.*, intercepting each loop iteration can introduce a significant overhead). We argue that turning a parallelizable loop into a method provides a simpler solution, which avoids many of the problems of pointcut for loops. The "inconvenience" of creating the loop method is insignificant compared with the problems previously mentioned. Although this approach may appear drastic at first, in reality, languages such as TBB and UPC follow the same strategy. Notwithstanding, exposing

[6] In this work, we use the same definition used by S. Chiba *et al.* in the paper [NC08] to define class refinement, namely: *"The term "refinement" is generally used in the context of formal methods. However, in this paper, it is used as a language mechanism for the extension to an existing class."*.

loops is the key to enable aspect modules to inject and compose different loop distributions (*e.g.*, combining static DM with dynamic SM loop distributions). B. Harbulot in [Har06] also recognizes the value of method loops for parallelism-related aspects in AspectJ.

In the context of parallelism, pointcuts to intercept arrays indexes/values [CC07] would be helpful, in our opinion, mostly in certain MPI parallelization patterns (*i.e.*, the mapping between the copy and original indices of an array scattered among processes). Possibly, such pointcut could also be used to implement a PGAS model. However, without any additional logic to remove unnecessary interception of indices (*e.g.*, compiler optimization), this approach would most likely introduce a huge overhead. In the context of parallelism, it is not clear to us the main practical advantages of such a fine grain join point. Moreover, using the same reasoning of the loop method, the accesses to the arrays to be intercepted can be performed through setters/getters methods, which, among others, have the benefit of hiding implementation details. Furthermore, Java provides collections (*e.g.*, ArrayList) with proper accessor methods that can be used instead of arrays. Regarding pointcuts for local variables [JMOA08], we fell that it could be useful in specific scenarios of SM parallelizations. However, in Java, the variables most relevant for parallelism are (usually) those belonging to object fields (more details are provided in Chapter 5). Therefore, the practical benefit of that pointcut, in the context of parallelism, is somewhat limited as well. Nonetheless, having such pointcut would be handy since, in some cases, turning local variables into fields can be considered as a code smell [Qub, FBB+99].

[XHG09, ACN09] proposed pointcuts to intercept synchronized blocks and entire blocks of code as join points, respectively. The former pointcut would not be useful in the context of our work since our main goal is to remove PRC from the base code, including synchronization blocks as well. If all the PRC are encapsulated into proper modules, there is no need to have a pointcut that intercepts synchronization blocks. In the latter pointcut to intercept a given code region, the programmer has to specify the beginning and end of that region (*e.g.*, between two method calls). This pointcut has several limitations, namely: - it can only intercept certain block regions due to dependencies between instructions; - the programmer has to define the region boundaries explicitly; - its fragility. For the most part, this pointcut tries to intercept a block of code that could have been a method, but it is not. Applying method refactoring is by far less complex than defining the limits of the code region to be intercepted. Moreover, typically these methods make the code more readable.

[AE06, CV13] propose the use of special annotations to make statements of the base code interceptable by AspectJ. These annotations, to some extent, work as hook methods but are more explicit in their intentions. This approach imposes the extension of the Java annotation API and the AspectJ pointcut language and requires the use of a precompiler to convert the annotations into AspectJ code. Some of the advantages pointed out in [AE06] is that annotations avoid the use of empty methods and the overhead of method calls. However, this annotation

approach also leads to several empty annotations interfaces, and nowadays these calls to empty method will likely be inlined. Finally, as acknowledged in [AE06] *"annotations require intrusive (nonoblivious) changes to the base program and do not completely modularize concerns."*.

Pointcut fragility [SG05] is a known problem in AOP and certainly would not be attenuated, most likely the opposite, with the introduction of finer grain and less stable join points. These join points would, most likely, require complex and fragile pointcuts that are hard to maintain [KAB07]. These issues may be some of the reasons why such pointcuts were never officially supported by AspectJ. Method call join point offers a more stable abstraction that can be used to inject the PRC. Furthermore, Java programmers are familiarized with the concept of extending behavior based on method calls (*e.g.*, with method overriding).

4.3.2 AOP and Parallelism-related concerns

AOP is the central theme of several publications and is used both in academic and enterprise environments [HJ09]. AspectGrid [Asp] and JEColi [EMR09], both developed at the University of Minho, are two examples of projects developed using AOP technology, specifically AspectJ. AspectGrid creates modules that adapt scientific applications to grid computing environments. This implementation uses AspectJ to provide distributed execution services, fault tolerance mechanisms [SM11], dynamic adaptation to resources [MS11], among others, through plugging the respective aspects. The works [PS+10] and [PSR13] used AspectJ to develop parallel versions of the JEColi framework where the parallelism-related code was moved into proper aspects.

[SLB02, TUSF03, CT04] and [Ban07] used AspectJ and AspectC++, respectively, to separate distribution concerns from domain code. For instance, [SLB02] used AspectJ to implement distribution and persistence requirements in the information system Health Watcher. The authors concluded that AspectJ provided a more robust implementation than with plain Java.

[ABVM10] introduced the concept of asynchronous *advice*, a technique to delay the execution of the code associated with a pointcut. The idea is similar to delay execution of certain blocks of code, which can also be used to introduce parallelism. [SMC06, Sob06] discusses the use of AspectJ to modularize and separate from the domain code different categories of PRC (*e.g.*, concurrency and partitioning). Based on that work, reusable libraries for concurrency [CSM06, CS07], partitioning [SCM07], and DM [GS12] were developed. [SM08, GS09] proposes a domain-specific language (DSL) [VDK02] that generates AspectJ code (needed for the PRC), hiding it from the user [GS12], and offers improved support for the composition of aspects. However, those DSL do not support AOP quantification, *"suffers a code bloat problem similar to that of C++ templates"* [SM08], and knowledge about AspectJ is still required when the developer needs to extend or add functionality. A survey of DSL for aspect languages can be found in [FDNT15] including the advantages and disadvantages of such an approach.

We build upon the knowledge from previous work but differing from it, by providing a complete framework of reusable aspects for clusters of multicores instead of only domain-specific [PS+10, PSR13], SM [CSM06, CS07], or DM [SCM07, GS12] PRC. The framework provides two libraries, based on well-known standards (*i.e.*, OpenMP and MPI), with competitive performance and easy composability – enabling the development of hybrid parallelizations. These libraries are enhanced with additional parallelism-related features (*e.g.*, object reduction and DM dynamic scheduler). The framework relies solely on AspectJ and Java without resorting to pointcut extensions or DSLs. The framework was developed to attenuate, from the perspective of the framework user, the composition problems of AspectJ, and the majority of its constructors rely on a smaller subset of AspectJ pointcut language. Furthermore, the parallelism-related constructors of our framework can be expressed with annotations and pointcuts. The former offers an elegant solution and the latter adds the flexibility necessary for sophisticated parallelism-related constructors. Unlike [CS07], our annotations are injected using ITDs, which avoid mixing these annotations with the domain code. Finally, the framework offers guidelines and design rules to reduce the likelihood of falling for the pitfalls of programming with AspectJ.

Chapter 5

Proposed Approach

5.1 Requirements

In the past few decades, the increase of hardware performance opened a world of new possibilities, and software complexity grew, hand in hand, with that of the hardware. Nowadays, parallel machines are mainstream and require new software programming techniques to manage the PRC. These PRC add additional complexity to the software development process since programmers have to be concerned about the implementation of the core (*i.e.*, domain) functionality and techniques to effectively exploit parallelism across multiple target platforms [MS13a].

Our primary goal is to provide an approach to reduce the complexity of PP while still achieving competitive performance. We believe that such an approach should promote the development of parallel applications with the following characteristics (core requirements):

1. Efficiency;
2. Modularity;
3. Pluggability;
4. Composability.

We want to develop a complete framework to support the development of parallel applications with the previously mentioned characteristics and, additionally, that framework should provide the following traits (additional requirements):

1. Usability;
2. Extensibility;
3. Reusability.

5.1.1 Core requirements

Efficiency

The first requirement is self-explanatory – the approach should enable the development of efficient parallel applications and efficiently support commonly used parallelism-related abstractions (*e.g.*, loop distributions), or otherwise, it would defeat the entire purpose of parallelism. Although efficiency is the primary goal, we do not believe that it should be achieved at all costs. We are willing to trade-off some performance if it leads to a decrease in the complexity of the software development process. Nonetheless, the parallelism-related constructors should have performance competitive with equivalent intrusive implementations (*e.g.*, OpenMP and MPI).

Modularity and Pluggability

A key concept, transversal to many sciences, is the reduction of the complexity of a task by splitting it into smaller manageable tasks. *"The technique of mastering complexity has been known since ancient times: divide et impera ..."* [Dij79]. Following the suggestion of D. Parnas [Par72a], a system should be divided into modules that encapsulate complex or likely-to-change design decisions. Moreover, those modules should be independent, interchangeable and cope with a well-defined responsibility (*i.e.*, *single responsibility principle* [Mar03]). Therefore, the PRC should be encapsulated into well-defined modules that can be easily changed (*i.e.*, modularity). For instance, synchronization and data replication are commonly used approaches to deal with data races. A well-designed modular system provides proper modules to, for example, encapsulate multiple strategies to deal with data races. Such a system would enable the users to tune the parallelization by changing, testing, and choosing the most appropriate module. In some cases using a module that provides synchronization capabilities is enough; in others, the use of a module that replicates data *per* thread is more appropriate.

The modularization of the PRC into well-defined modules that perform a specific task promotes the development of parallel applications with higher *cohesion*[1] modules. Consequently, providing structure and abstraction, promoting correctness (*i.e.*, it is easy to debug smaller and highly cohesive modules) and enabling independent development (*i.e.*, programmers can work independently on different modules).

The parallelism-related modules should not be tightly-coupled with the base code and *vice-versa* (*i.e.*, reducing the coupling between parallelism- and domain- related modules). To achieve that goal, a common ground between the base code and the parallelism-related modules should be established, where both communicate through a proper interface. Furthermore, the parallelism-related modules should be (un)plugged surreptitiously from the base code, ideally, without

[1] Measures the *"degree to which the elements inside a module belong together"*[YC79]. The higher the module cohesion, the better.

requiring any change to it (pluggability[2] [SCM07, GS09, MS11]). The base code is oblivious to the presence of the pluggable modules (obliviousness). Hence, these parallelism modules are optional, meaning that the base code works correctly without them. Pluggable modules facilitate the use of the *design for change* principle introduced by D. Parnas [Par79], that advocates for the development of software that can be extended with new functionality in a non-invasive fashion [Ape07]. Pluggability also helps with correctness, since the programmer can effortlessly (dis)connect different PRC by (un)plugging the correspondent modules which facilitates debugging. Furthermore, modularity and pluggability enable the support of sequential semantic[3][MRR12]; the user can unplug all the parallelism-related modules, and the application will correctly execute sequentially. By providing a modular system with pluggability capabilities, we can further aim for composability.

Composability

Composability is highly dependent on modularity. Initially, a big system should be divided into smaller modules, which can later be composed/combined, in different orders, to provide solutions to a given problem. It is common to combine multiple PP paradigms (*e.g.*, SM + DM) to exploit hybrid environments as clusters of multicore machines, which increases the complexity of PP even more. Therefore, it is paramount that modules, from different PP paradigms, can be composed with a given base code with minimal intervention from the programmer. Otherwise, for instance, to distribute the iterations of a loop across threads from multiple processes the programmer would have to code the entire logic behind it. Such an approach besides being inflexible to changes is also monotonous and error-prone. Whenever the programmer wants to test different combinations of distributions among threads and processes, the code has to be adapted again. Oppositely, in a system that encapsulates these low-level details into modules which can be seamlessly composed with the base code, the programmer would only be responsible for choosing the type of SM and DM loop distribution to be combined. The entire compositional process of the two loop distributions would be performed in the background by the system. Thus, this reduces the complexity and likelihood of bugs and improves software maintenance. Once proven that the SM and DM loop distributions can be correctly composed with the base code, regardless of the loop range, this heuristic can be reused reliably in different applications. The same reasoning can be applied to other combinations of parallelism-related constructors of hybrid parallelizations.

The composition should not be only attained among different PP paradigms, but also within those paradigms themselves. More than just being able to compose modules from multiple PP paradigms, the user should also be able to compose the (smaller) modules from a specific PP

[2] In the context of this thesis, we defined pluggability as the ability to add and remove modules from a base code in an oblivious manner.

[3] Means that a program maintains the same semantics regardless of running sequentially or in parallel [MRR12].

paradigm (*e.g.*, SM). By providing a panoply of composable modules enhanced with pluggability properties, the system promotes a flexible composition approach. Consequently, to parallelize an application, the user is not forced to apply the same rigid composition structure (*e.g.*, having to add always the same paradigm-related module). Instead, the user has the flexibility of adding multiples, a single or even no parallelism-related module(s).

Consequential properties from the core requirements

After guaranteeing that the framework enables the development of parallel applications with the desirable properties (*e.g.*, modularity and pluggability) we argue in favor of incremental development [Dij79, BSR04, Sob06] to tackle the difficulty of PP. The programmer should break-down the PP complexity by first separating between domain- and parallelism- related concerns. Initially, the programmer should focus mainly on implementing the DRC in a well-designed, maintainable and understandable manner. The programmer should first guarantee the application correctness and only then focus on performance. Afterward, parallelism can be incrementally added to the base code to speed up the application. The phase of adding parallelism itself should be divided into multiple development steps as well, such as finding the hotspots to insert parallelism, dealing with data dependencies, task granularity, load balancing and so on. It is preferable to have an application that does what it was meant to do, even if with a lower performance, then the other way around. With incremental and structured development not only does the programmer break down the complexity of PP but also makes it easier to debug. The programmer can separately and gradually debug different phases of the software development and narrow down the sources of bugs and hazards.

Languages such as OpenMP and MPI[4] provide abstractions that can be transparently mapped to different architectures. This philosophy helps to reduce the complexity of PP and also promises performance portability. The idea is that programmers can express parallelism for different hardware (*e.g.*, CPU and GPU) using a single syntax and the compiler generates efficient code for each target platform. However, achieving performance portability for different platforms with this approach can be very hard. Most likely the user will have to make, at a certain point, versions which are tuned to specific target platforms. For instance, a parallel *for* with a static block distribution will likely be suitable to a system with multicores since it is cache friendly. However, it will suffer a performance penalty in GPUs because of memory coalesce [FPS15] problems. Modularity and pluggability in conjunction with composability, facilitate performance portability. Different sets of modules can be swapped to meet the performance constraints of the target architectures and composed to exploit hybrid environments. Moreover, the user can change the parallelization based on the target platform (*e.g.*, changing from a SM to a DM parallelization) without adjusting the base code.

[4]With the SM and one-side communication features.

5.1.2 Additional requirements

Usability

Usability can be hard to measure especially because this depends on the users' background. For instance, a C++ programmer (typically) will learn TBB faster than a FORTRAN programmer would. Nevertheless, the framework should try to minimize the set of new concepts that a programmer with prior knowledge of OO programming, multithreading and messaging-passing should learn. Furthermore, methodologies should be provided to the user to ease not only the learning curve of the framework but also the task of PP.

Popular programming systems such as OpenMP and TBB offer high-level abstractions to reduce the complexity of PP. Those abstractions hide some of the low-level details of PP (coined by J. Reinders as the *"assembly of parallelism"* [Rei07]). Accordingly, it is also critical for the reduction of the complexity of PP that the framework hides the low-level details of parallelism from the programmer. The *average* programmer should not have to worry, for instance, about implementing schedulers and thread pools from scratch but focus on the high-level details, instead. Hence, the framework should provide commonly used high-level parallelism-related abstractions so the programmer can concentrate on the essential.

Extensibility

Ideally, the framework would provide all the constructors that a user would ever need, however, that is an unrealistic scenario. Therefore, the framework should provide configurable constructors to adapt to specific needs. As previously mentioned, the framework should provide high-level constructors that hide most of the low-level details. Nonetheless, we acknowledge that in some scenarios, (*advanced*) users should be allowed to access finer grain constructors to build their tuned versions (*e.g.*, a customized loop distribution). Users might frequently encounter the same domain-specific parallelism-related constructors that by default are not provided by the framework. Hence, users should be able to extend the framework with their own set of constructors.

Reusability

The framework can achieve reusability by providing a set of reusable libraries, one *per* PP paradigm, with commonly used PP abstractions. Furthermore, the user should be able to reuse his/her customized modules in different applications. Based on the core requirements, one question that arises is whether it is possible to create reusable libraries, which can be used in a base code oblivious to their presence.

5.2 Conceptual overview

Modularity and pluggability are two essential properties used in this work to achieve composability, which consequently enables incremental and independent development. As pointed out by S. Apel, *"A module is a device to implement a concern and modularity is a consequence of separation of concerns."*[Ape07]. Unfortunately, the traditional approach of mixing PRC with DRC limits modular software development and has strong implications for software maintenance and evolution. By tightly coupling the PRC with the DRC, in addition to obscuring the domain code, it also makes it harder for the programmer to reason and isolate parts of the code. Consequently, changing either the DRC or PRC without them affecting each other also becomes harder. Annotations-based approaches and OO mechanisms (*e.g.*, class extension) offer cleaner solutions but do not completely avoid the mixing of domain- and parallelism- related concerns.

To deal with the problems previously mentioned we introduce *parallelism layers*, an idea built upon the concepts of class refinement [NC08] and mixin layers [SB02]. Class refinement is inspired by well-known OO concepts (*i.e.*, class extension and method overriding), making it familiar to OO programmers. With class refinement, one can add functionality and state to a base class in a similar way to subclassing. Moreover, a class refinement can, just like a subclass, access state and behavior from the base class. Nevertheless, class refinement, unlike subclassing, does not alter the class hierarchy (*i.e.*, extension), which avoids the scalability problems of OO inheritance (*e.g.*, *exploding class hierarchy*) [Kuc04]. These class refinements can be seen as program transformations that can add additional behavior which the domain code is oblivious to. Those simple, yet powerful, characteristics make class refinement suitable to solve the tangling problems (detailed in Section 3.2) in a modular and pluggable manner.

Even though class refinement can be used to separate PRC from DRC, and consequently solve the tangling problem, the PRC will still be spread across different class refinements, which leads to scattering issues. Furthermore, with class refinement, multiple refinements can be applied to the same class, but the same refinement cannot be applied to multiple classes. A layer overcomes those problems by grouping related refinements together and by enhancing class refinement with a mechanism to apply the same refinement to multiple classes (*e.g.*, a mixin).

We aim to inject parallelism into applications through a framework based on the concept of parallelism layers. These layers encapsulate a set of class refinements that can be non-invasively attached to specific points of the base code (*i.e.*, *extension points*[5]). These class refinements work as code transformations, which the base code is oblivious to. Modularity is achieved by encapsulating those code transformations into proper modules that will, in turn, be aggregated into layers. Each layer represents a specific PP paradigm (*e.g.*, SM Layer and DM Layer in Figure 5.1) and each module represents a certain kind of code transformation (*e.g.*, Parallel region). Pluggability – an intrinsic property of class refinements – comes from the combination of modularity and the fact that layers can be seamlessly added/removed to/from the base code.

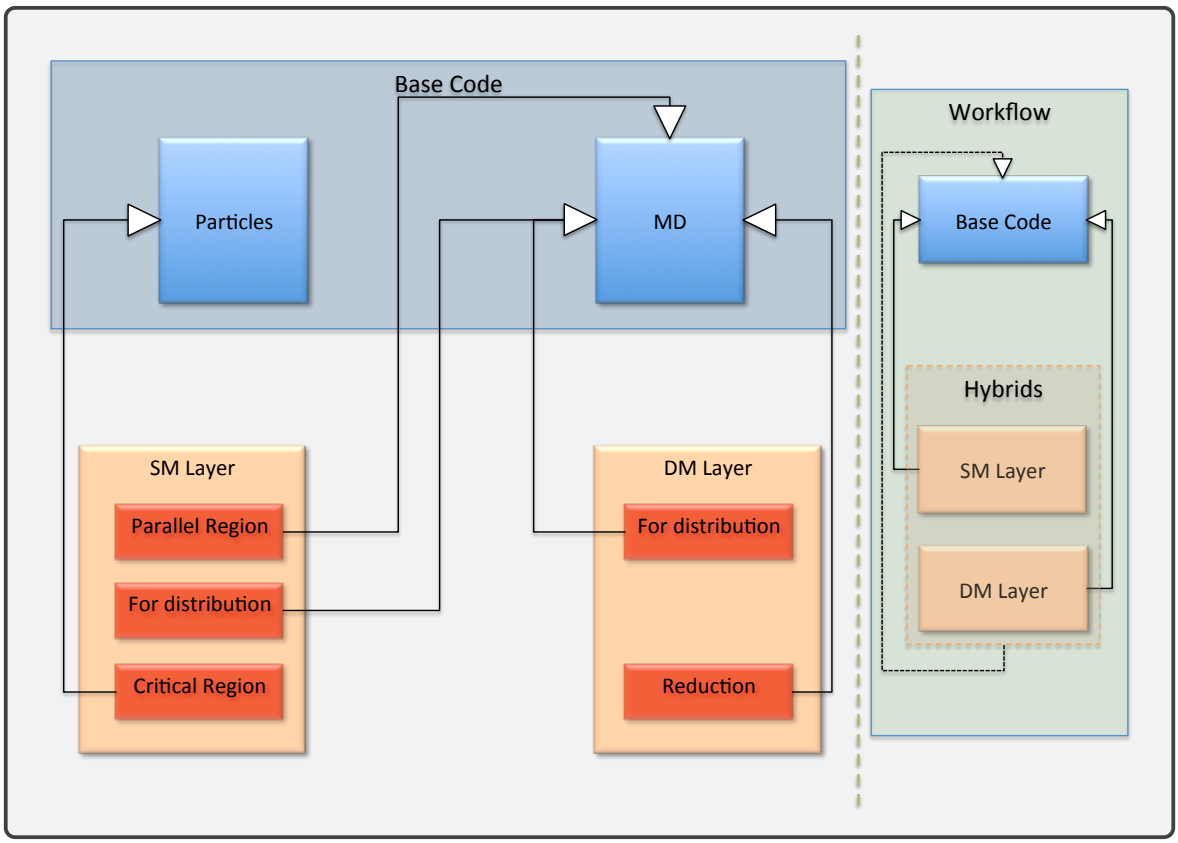


Figure 5.1: A high-level overview of the parallelism layers approach.

Figure 5.1 shows a high-level overview of the parallelization of a program with two base classes using our approach (for simplicity their methods/state are omitted). The green, blue, orange and red rectangles represent the workflow of using the parallelism layers to parallelize applications, the base code, the parallelism layers, and the parallelism-related modules, respectively. The parallelization of an application is materialized by adding one or more layers, each composed of one or more modules that in turn can encapsulate one or more code transformations.

[5]Points of the base code that are used to inject the PRC using our framework are labeled by us as *extension points*.

With our approach, the programmer starts by developing and composing the SM layer[6] with the base code; inside that layer, the programmer includes the modules to be applied. In the example of Figure 5.1, the programmer intends to inject a parallel region, a *for* distribution and a critical region into extension points of the classes *MD* and *Particles*. After guaranteeing the correctness of the SM parallelization, the programmer can then develop the DM layer using the same process. At the end of the workflow, the SM and DM layers can both be composed with the base code to provide a hybrid solution. Since the PRC are not tangled with the DRC, the programmer only has to unplug the undesirable layer(s) to execute the base code without parallelism or with only one PP paradigm. The modularity and pluggability, and consequently composability, properties of the parallelism layers make this abstraction perfect for the quick prototyping of different parallelism-related strategies. For instance, the programmer can test different loop distributions and lock strategies just by swapping the appropriate modules or by creating new layers with different parallelizations. Furthermore, with these properties, one can aim for performance portability (*i.e.*, layers can be (un)plugged based on the target architecture).

Modularity and pluggability do not come for *free* – to have both properties in addition to offering modules that encapsulate the PRC, the parallelism layers should also provide an explicit interface. The designing of reusable and pluggable parallelism-related modules that can be applied to different base codes without modifying either side is a surreal task. Consequently, the parallelism and base code modules have to agree on some interface, which is a standard protocol in some OO languages (*i.e.*, different modules should communicate via interfaces [Par72b]). The units of extensibility of our approach are method calls and accesses to fields, which are the parallelism layers' expectable interface. Coming back to the example of Figure 5.1, in that case, the programmer would have to ensure that the points to be extended in the *MD* and *Particles* classes are method calls. The next subsection presents a more detailed illustration of this process. Limiting extensible points to only methods and fields accesses restricts the parallelism interface, which eases its use by programmers, promotes more structure and reduces complexity. So far the creation of the parallelism layers has only required the materialization of concepts commonly known to OO programmers. Analogously, the creation of the method extension points and the selection of PRC modules follows the same reasoning of creating a method to be overridden in the future and creating classes in OO, respectively.

[6] Technically, the programmer could have also started with the DM layer.

5.3 Description

AOP provides an excellent background for materializing the concept of parallelism layers. AOP has, among others, mechanisms that can: - emulate class refinements; - group related class refinements together; - apply the same class refinement to multiple classes. Hence, avoiding the tangling and scattering problems.

We have used AspectJ and its mechanisms, such as *advices*, aspects, and pointcuts, to materialize the concepts of parallelism layer. From an empirical point of view, *advices* and aspects can be seen as mechanisms capable of emulating class refinements and encapsulating sets of related class refinements, respectively. The pointcuts establish the mapping between refinements and the (base) classes where these refinements will be applied.

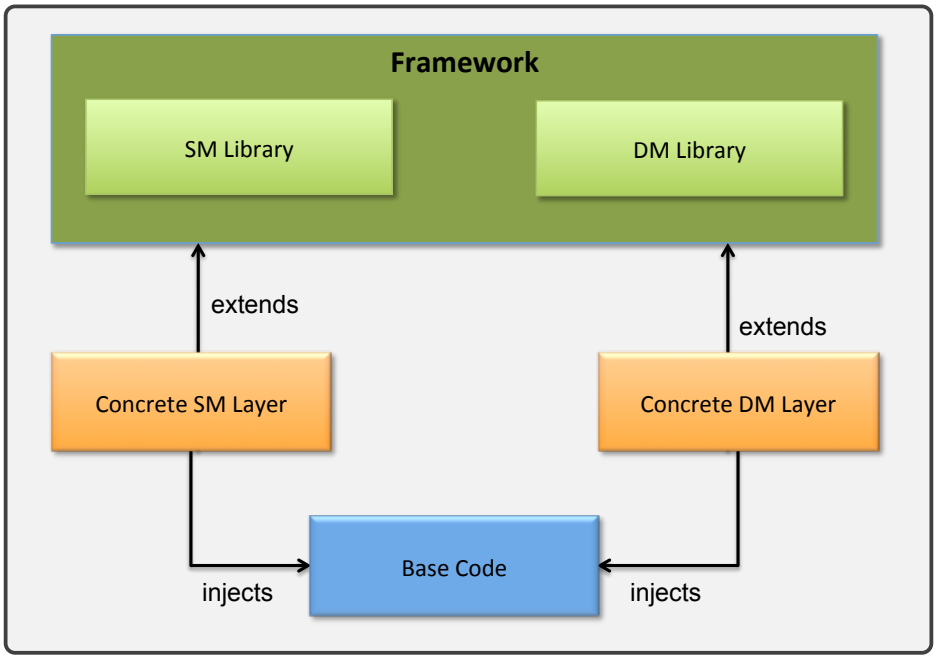


Figure 5.2: The three main components for those developing parallel applications.

From programmer's point of view, our materialization of parallelism layers with AspectJ can be divided into three main components (illustrated in Figure 5.2), namely the libraries of aspects, the concrete aspect layers, and the base code. All of these three components work in a decoupled manner. Accordingly, our libraries of aspects were designed to be independent of the join points of a specific base code. The libraries are composed of abstract aspects that encapsulate behavior and state, transversal to their sub-aspects, and also composed of abstract pointcuts without explicitly defined join points. To create a parallelism layer with these libraries, the programmer extends the abstract aspect layer of the corresponding library (*e.g.*, SM) into a concrete one. Inside these concrete layers, the programmer selects the parallelism-related

modules (*e.g.*, parallel region). This selection process consists of the creation of (concrete) inner aspects that extend the corresponding (abstract) ones from the library of aspects (*e.g.*, abstract parallel region). These concrete aspects encapsulate state and behavior specific to the base code. Finally, the mapping between the abstract pointcuts and the join points to be intercepted is also defined in the concrete aspects. Alternatively, instead of creating the inner aspects, the programmer can use inter-type declarations (ITDs) combined with our annotations and insert them directly into the concrete layer. By using ITDs, these annotations are not added (intrusively) into the base code, but rather centralized in the concrete layer. The customization of the parallelism-related constructors (*e.g.*, *chunk=2*) with a pointcut approach is performed through method overriding, whereas with annotations it is done using the annotation parameters. Unfortunately, annotations are not as flexible as pointcuts. Hence, their use is limited to a set of constructors.

The concrete aspect layer works as a bridge between the library's core and the base code, resembling the use of XML configuration files in the Spring AOP framework [Lad09]. Finally, from the application's point of view, sometimes it is necessary to expose join points (*i.e.*, extension points) using our design rules (explained in more detail in Section 5.3.3). This kind of approach is known in requirements engineering as *scaffolding* [CGS+09, Byn11]. In our framework, the PRC are injected into method calls and fields. Hence, our design rules are mainly about turning join points from base code into points (*e.g.*, method calls) that can be interceptable by our framework constructors. Additionally, in some cases, it is expected that extra context is provided, for example, the *for* loop body range (*i.e.*, begin, end and step).

Programmers can choose from a panoply of parallelism-related code transformations. The framework provides several commonly used parallelism-related constructors for both SM and DM, most of them based on the OpenMP and MPI standards, respectively. Nevertheless, just as with OpenMP and MPI parallelizations, sometimes different or more tuned code transformations might be needed. Typically, such a situation would not be a problem because the programmer could always use explicit calls to the API and customize their code transformations (*e.g.*, a handmade loop distribution). However, our framework philosophy is to not insert PRC directly into the base code. Therefore, our framework provides specialized modules to support additional code transformations, such as *injecting* behavior before and after a method call [KAB07, ea09], manipulating arguments of a method, and creating user-defined loop distributions. In case these additional transformations are still not enough, *advanced users* can always extend the libraries with their own aspects, pointcuts, and *advices* to express their customized code transformations.

5.3.1 Illustrative example

This section uses the MD example of Figure 5.1 to show how the framework can be used to parallelize applications. This example is based on the MD of the *Java Grande benchmark suite* [SBO01] and it is also exploited in more detail in Section 6.

```
1  public class MD {
2    private final Particles particles;
3    private double epot, vir;
4    private int interactions;
5      ...
6    protected void updateControlVar(double e, double v, int i){
7      this.epot += e;
8      this.vir += v;
9      this.interactions += i;
10   }
11   protected void cicleForcesNewtonsLaw(){
12     epot = vir = 0.0;
13     particles.calculate_force(this);
14   }
15   ...
16 }
```

Figure 5.3: MD - Code snippet of the *MD* class.

```
1  public class Particles {
2    private final double fx[],fy[],fz[]; // forces
3    ...
4    Particles(final int size){
5      fx = new double [size];
6      ...
7    }
8    protected void calculate_force(final MD md){
9      for(int i = 0; i < totalParticles; i++)
10       forceNewtonsLaw(md, i);
11   }
12   ...
13   private void forceNewtonsLaw(final MD md, int pA){
14     ...
15     /** Save MD local variables values **/
16     double epot = 0.0, vir = 0.0;
17     int interactions = 0;
18     for (int pB = pA + 1; pB < totalParticles; pB++){
19       ...
20       if(/** pB inside the radius of pA**/){
21         ...
22         /** Calculating thirds Newton's law */
23         fx[pB] -= tmpFx;
24         fy[pB] -= tmpFy;
25         fz[pB] -= tmpFz;
26         ... // update local epot, vir interactions
27       }
28     }
29     /** Update of the force of Particle A **/
30     fx[pA] += fxAcc;
31     fy[pA] += fyAcc;
32     fz[pA] += fzAcc;
33     /** MD control variables actualization */
34     md.updateControlVar(epot, vir, interactions);
35   }
36 }
```

Figure 5.4: MD - Code snippet of the *Particles* class.

Figures 5.3 and 5.4 present the most relevant code of the two main classes of the MD case study, *MD* and *Particles*, respectively. For now, it is only relevant to know that the SM parallelization adds a parallel region and a *for* distribution to the *calculate_force* method call in the class *MD* (line 13 of Figure 5.3) and critical regions to the class *Particles*. These critical regions are added to the update of forces between two particles (inside the *forceNewtonsLaw* method) and to the call of the *updateControlVar* method (line 34 of Figure 5.4). All those class refinements (*e.g.*, *for* distribution) were illustrated previously in Figure 5.1.

After having identified the constructors to use, it is time to apply the necessary design rules (*e.g.*, method refactoring). The programmer starts with the introduction of the *for method* design rule (shown in Figures 5.5 and 5.6). Additionally, the programmer merely has to expose the loop range of the *for* loop of the *calculate_force* method (line 9 of Figure 5.4).

```
11 protected void cicleForcesNewtonsLaw(){
12   epot = vir = 0.0;
13   particles.calculate_force(0, particles.getTotalParticles(), 1, this);
14 }
```

Figure 5.5: MD - The *for* design rule in the *MD* class.

```
8 protected void calculate_force(int begin, int end, int step, final MD md) {
9   for(int i = begin; i < end; i += step)
10    forceNewtonsLaw(md, i);
11 }
```

Figure 5.6: MD - The *for* design rule in the *Particles* class.

After having applied the *for method* design rule, the programmer develops the SM layer by first creating an aspect that extends the abstract SM layer (line 1 of Figure 5.7). Afterward, the programmer creates concrete aspects that extend the parallel region (line 5) and the *for* distribution (line 9) abstract aspects, chooses the desirable code transformations (*i.e.*, *parallel* and *for_static*) and identifies the appropriate extension points (lines 6 and 10). The programmer also customizes some *for* constructor options (lines 12 and 14).

```
1  aspect MD_SM extends SM_Layer
2  {
3    pointcut hotSpot(): call(protected void Particles.calculate_force(int,int,int,MD));
4
5    static aspect parallel_Region extends Sm_Parallel{
6      pointcut parallel() : hotSpot();
7    }
8
9    static aspect worksharing extends Sm_For{
10     pointcut for_static() : hotSpot();
11
12      @Override public boolean no_wait()  {return true;}
13
14      @Override public int chunk ()       {return 1;}
15   }
16 }
```

Figure 5.7: MD - The parallel *for* constructor with pointcuts.

After having applied the parallel *for* constructor, the programmer adds the critical regions. Those regions required the use of the *method* design rule (shown in lines 26 and 30 of Figure 5.8) over the update of the forces. This design rule encapsulated lines 23 to 25 and 30 to 32 of Figure 5.4 into a method (lines 35 to 39 of Figure 5.8). Finally, there is no need for additional design rules to apply a critical region over the *updateControlVar* method call.

```
25   ...      /** Calculating thirds Newton's law */
26          forceUpdate(pB, -tmpFx, -tmpFy, -tmpFz);  // <--- Design Rule
27        }
28      }
29      /** Update of the force of Particle A **/
30      forceUpdate(pA, fxAcc, fyAcc, fzAcc);       // <--- Design Rule
31      /** MD control variables actualization */
32      md.updateControlVar(epot, vir, interactions);
33      }
34
35   ... void forceUpdate(int p, double tFx, double tFy, double tFz){  // <--- Design Rule
36      fx[p] += tFx;
37      fy[p] += tFy;
38      fz[p] += tFz;
39   }
```

Figure 5.8: MD - Applying the *method* design rule to use the *critical* constructor.

After the design rules, the programmer adds to the SM layer a concrete aspect that extends the *synchronization* abstract aspect and selects the appropriate parallelism-related constructor (Figure 5.9). OpenMP offers the possibility of naming critical regions so that threads can differentiate among those regions. All nameless critical regions are treated as being the same one and consequently using the same internal lock. Rather than the traditional OpenMP *critical* alike constructor, the programmer used a slightly different variation named *critical_method*. This constructor works as a *synchronized* method in Java. Thus, unlike with OpenMP, there is no need to name the critical regions so that threads can differentiate between those regions.

```
17 static aspect criticalParticles extends Sm_Sychronization {
18   pointcut critical_method() : call(...void ...forceUpdate(...)) ||
19                                call(...void ...updateControlVar(...));
20 }
```

Figure 5.9: MD - Pointcut of the *critical* constructor.

The same parallelization with OpenMP (Figure 5.10) required less programming effort than using our framework. However, this will not always be the case as we will present in Section 6. Nonetheless, our framework also offers annotations to inject the PRC (Figure 5.11). Besides inserting the OpenMP annotations, the programmer also had to encapsulate the update of forces within brackets (lines 12, 16, 22 and 26 of Figure 5.10). Furthermore, in addition to the code tangling cause by mixing PRC and DRC, the *critical(criticalForces)* constructor was applied twice, resulting in code duplication (line 11 and 21). The encapsulation into brackets and the code duplication could have been avoided by simply creating a method out of the update of forces (as it was done in our approach).

```
1   ...
2   protected void calculate_force(final MD md){
3     #pragma omp parallel for schedule (static,1)
4     for(int i = 0; i < totalParticles; i++)
5       forceNewtonsLaw(md, i);
6   }
7   ...
8   private void forceNewtonsLaw(final MD md, int pA){
9         ...
10        /** Calculating thirds Newton's law */
11        #pragma omp critical(criticalForces)
12        {
13          fx[pB] -= tmpFx;
14          fy[pB] -= tmpFy;
15          fz[pB] -= tmpFz;
16        }
17        ... // update local epot, vir interactions
18      }
19    }
20    /** Update of the force of Particle A **/
21    #pragma omp critical(criticalForces)
22    {
23      fx[pA] += fxAcc;
24      fy[pA] += fyAcc;
25      fz[pA] += fzAcc;
26    }
27    /** MD control variables actualization */
28    #pragma omp critical(criticalMDvariables)
29    md.updateControlVar(epot, vir, interactions);
30  }
```

Figure 5.10: MD - The OpenMP parallelization with the use of the critical constructor.

```
1 aspect MD_SM extends SM_Layer {
2   declare @method: ...Particles.calculate_force(...): @Parallel_for_static(chunk=1);
3   declare @method: ...Particles.forceUpdate(...)     : @CriticalMethod;
4   declare @method: ...MD.updateControlVar(...)       : @CriticalMethod;
5 }
```

Figure 5.11: MD - The parallel *for* constructor with annotations.

In the DM parallelization, the programmer wants a *for* that distributes the iterations of the force calculation among processes. Furthermore, after the entire force calculation process, the fields *epot*, *vir* and *interactions* from the *MD* object, along with the forces from the *Particles* object, should be reduced among processes. As with the SM layer workflow, initially, the programmer creates an aspect that extends the abstract DM layer. Afterward, the programmer creates a concrete aspect that extends the *for* distribution abstract aspect (line 3 of Figure 5.12) and identifies the appropriate extension point (line 4). This time, no *for method* design rule is needed since the programmer can reuse the one from the SM layer workflow. Finally, the programmer performs the data reduction by creating an aspect that extends the appropriate module from the DM library (line 9) along with defining the data to be reduced, and after which method. The communication routines and their data are specified in the *data* method (lines 12 to 17); detailed information about those constructors is provided on Sections 5.3.5 and 5.4.3.

```
1  aspect MD_DM extends DM_Layer {
2
3    static aspect workSharing extends Dm_For {
4      pointcut for_static() : call(...void calculate_force(..));
5
6      @Override public int chunk(){return 1;}
7    }
8
9    static aspect ReduceVariables extends Dm_Comm <MD>{
10     pointcut comm_after() : call(...cicleForcesNewtonsLaw());
11
12     @Override public void data (MD md){
13       ...
14       md.setVir             (allReduce(md.getVir() , Reduction_DM_OP.SUM));
15       ...
16       allReduce             (md.getParticles().getFx(), Reduction_DM_OP.SUM);
17     }
18   }
19
20   static aspect InitVariables extends Dm_Injection <MD> {
21     pointcut inject_code_before() : call(...cicleForcesNewtonsLaw());
22
23     @Override public void inject_code_before (MD md) {
24       if(!isMaster())  md.setInteractions(0);
25     }
26   }
27 }
```

Figure 5.12: MD - The DM initial parallelism layer.

After executing the base code with the DM layer, the programmer noticed that something was missing (*i.e.*, a variable initialization[7]) that could not be provided by the framework constructors. Nevertheless, the programmer was able to create and add the additional missing logic to the base code (lines 20 to 26).

To create a hybrid parallelization, in this example, the programmer only needs to add both the SM and DM layers to the build. Behind the scenes, as soon as the MD is executed the DM layer creates multiple instances of it (one *per* process) and ensures that the MPI processes are correctly initialized/terminated. Before and after the execution of the *cicleForcesNewtonsLaw* method, the DM layer initializes the field *interaction* and reduces (across processes) some fields from the *MD* and *Particles* classes, respectively. Finally, before the execution of the *calculate_force* method:

1. the SM layer wraps it around within a parallel region;
2. the DM layer statically distributes the iterations of its loop among processes;
3. the SM layer statically distributes the iterations assigned to a process among its threads.

The SM layer also ensures that the methods *forceUpdate* and *updateControlVar* are executed with mutual exclusion, within each process.

[7]That initialization is necessary because the *interaction* variable is reduced (using the sum operation) across processes. The same initialization was not applied to the variables *epot* and *vir* because their initialization is already done in the base code (line 12 of Figure 5.3).

5.3.2 Overview of the Framework

Figure 5.13 presents a more detailed view of our approach and framework components. The light-green and orange rectangles represent abstract and concrete parallelism layers, respectively. The red rectangles represent the available class refinements (*i.e.*, code transformations), and the light-blue rectangles within the concrete modules represent the points from the base code where the code transformations are applied (*i.e.*, extension points). The (abstract/concrete) layers, the code transformations, and the specification of the extension points are performed with AOP using (abstract/concrete) aspects, *advices*, and pointcuts (or annotations with ITDs), respectively.

For simplicity reasons, we omitted from Figure 5.13 several details of the framework, such as all its SM and DM abstract modules (*e.g.*, *for* distribution) and their code transformations (*e.g.*, *static for* and *dynamic for*). Nonetheless, the "Module X" and "Module Y" illustrate the coexistence of several parallelism modules, each implementing a different set of code transformations (labeled in the Figure 5.13 as "C.T.").

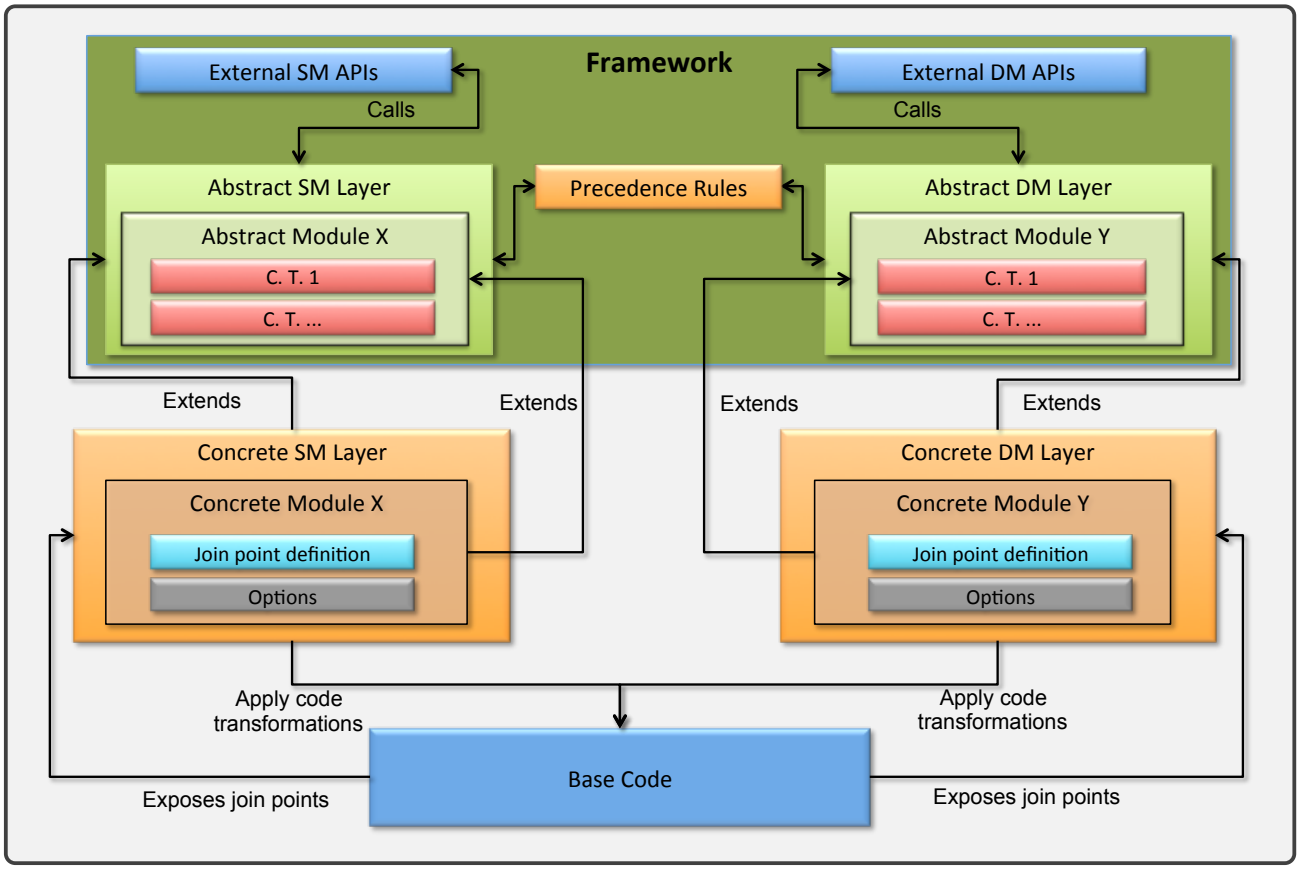


Figure 5.13: Framework and approach detailed view.

The framework is composed of two libraries of aspects one for each PP paradigm, namely SM and DM – represented by "Abstract SM Layer" and "Abstract DM Layer" in Figure 5.13. The SM (AOmpLib [MS13a, MS17]) and DM (AOdmLib [MS17]) libraries are influenced by the OpenMP and MPI standards, respectively. On top of the constructors defined in the OpenMP and MPI standards, both libraries of aspects also provide additional higher-level constructors (*e.g.*, DM *for* distribution). This combination of commonly known standards, enhanced with higher-level constructors on top of them, increases the usability, reliability, and acceptability of the framework, and reduces the complexity of developing parallel applications.

With AOmpLib, as with OpenMP, whenever a thread reaches a parallel region, it requests a new team of threads from the initial pool and becomes the master of this team [MS17]. As with OpenMP, AOmpLib focuses on loop-level parallelism and task parallelism. AOdmLib runs as many instances of the base code as the number of processes requested using the SPMD execution model of MPI. Similar to OpenMP and MPI, our libraries do not automatically check for data dependencies, race conditions or deadlocks. An overview of AOmpLib and AOdmLib is presented in Sections 5.3.4 and 5.3.5, respectively. Additionally, Sections 5.4.2 and 5.4.3 present some of the technical details about the implementation of AOmpLib and AOdmLib, respectively.

Modularity

From the framework development point of view, AOP enables the modularization of the PRC through their encapsulation into aspects. From the perspective of the framework users, AOP enables the parallelization of their applications in a modular manner by allowing them to map the framework modules to the points in the base code to inject the parallelism. In our framework, each library contains a set of specialized modules to deal with a specific PRC. For instance, AOmpLib has specialized modules for loop distributions (*e.g.*, static *for*), synchronization mechanisms and many others. Each of these modules contains all the logic (*e.g.*, *advices*, pointcuts and methods) related to a certain kind of code transformation (*e.g.*, loop distribution). By breaking down the libraries into several smaller aspects, we not only reduced the complexity of the libraries themselves, but also increased the modularity of the framework and the *cohesion* [YC79] of its modules. Consequently, promoting the development of modular parallel applications, users have a panoply of parallelism-related modules to choose from, each tackling a specific PRC. Moreover, having smaller parallelism-related modules also facilitates the specification of aspect precedences, which has consequences for the composition of these modules with the base code. However, the downside of this approach is that the user has to do more programming during the implementation of the concrete layers. For instance, to use a parallel *for* constructor the user has to extend two aspects and define two pointcuts (*e.g.*, Figure 5.7) instead of one aspect and one pointcut. Nonetheless, those extra lines of code are a small price to pay for modularity; otherwise we could have added all PRC into a single aspect.

Pluggability

One of the key properties of the parallelism layers is the ability to (un)plug modules without having to change the semantics of the base code. This property enables sequential semantics when all the parallelism-related modules are unplugged and facilitates the parallelization tuning process (*i.e.*, the customization of parallelism), among others. The first step towards pluggable modules was achieved by encapsulating the PRC into proper modules (*i.e.*, modularity). With AOP one can add/remove aspects to/from the base code in an oblivious manner as long as the join points to be intercepted are part of the AOP join point model. From that model, however, the parallelism layers are only interested in method calls and fields to inject the parallelism-related constructors. Hence, the base code might have to be refactor to expose desirable join points for the parallelism. In our approach these join points are exposed by using structural patterns [Nag06, Byn11] namely, *extract method refactoring* [Fow99], and turning primitive local variables into objects or fields (*i.e.*, *extract field refactoring*). During the description of the framework requirements (Section 5.1) the following question arose:

"Based on the core requirements, one question that arises is whether it is possible to create reusable libraries, which can be used in a base code oblivious to their presence."

Our libraries, to be reusable, have to be independent (*i.e.*, not change) from the parallelization of the base code. Therefore, these libraries should be able to intercept the join points and abstract them from the concrete ones of a given application. In our framework, each library is composed of a hierarchical *tree* of abstract aspects (*nodes*) and concrete aspects (*leaves*) to avoid being tightly-coupled with specific join points of a given base code. These abstract aspects contain *advices* and (abstract) pointcuts[8] [HU01, HS03], which are also complemented with the use of marker interfaces [HC02, HS03], parameterized types [HU01, HE08], and annotations [Lad09]. For each base code, the abstract aspects are extended by concrete ones that contain the pointcuts/annotations, state, and behavior defined by the user.

To make the base code oblivious to the presence of the libraries' parallelism modules, the base code has to be prepared in a way that exposes the desirable join points to be injected with parallelism without changing the base code semantics (*i.e.*, by applying our design rules). After defining the expectable pointcut interface and structuring the base code accordingly, the concrete layer comes into play. In our approach, a concrete layer represents an entity that materializes the relationship of the pair base code and parallelism layer. This entity holds the mappings between the parallelism-related code transformations (provided by the framework) and the join points from the base code to be intercepted. The concrete layers exist so that neither the base codes nor the libraries have to change their implementations to adapt to a specific parallelization.

[8] By abstract pointcuts, we include those that use the *abstract* keyword as well as those that do not have a concrete implementation (*e.g.*, *pointcut for_static (int, int, int);*)) and can be later overridden.

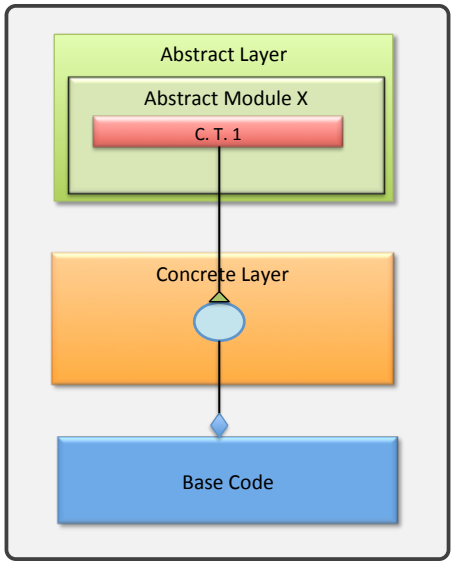


Figure 5.14: The mapping between parallelism-related constructors and the base code join points.

Figure 5.14 shows the relationship among the parallelism modules, the (newly created) entity where the programmer expresses the intended parallelism (Concrete Layer) and the base code. The parallelism modules contain the actual implementation of the code transformations but without the specification of the join points where these transformations will be applied. Hence, the code transformations are independent of the base code. Nevertheless, these code transformations expose details about the type of join points supported (*e.g.*, a method call) and the expected context (*e.g.*, a specific argument of the intercepted method). These code transformations can be seen as black boxes that require join points that satisfy their requirements (represented in Figure 5.14 by the green triangle). Similar reasoning is used for the base code side as well. To apply a specific code transformation in a join point of the base code this point needs to respect the requirements of the code transformation (represented in Figure 5.14 by a blue diamond). Consequently, sometimes our design rules have to be applied to base code points to turn them into points which are interceptable by our framework. After applying the design rules, the programmer expresses the desired parallelism (represented in Figure 5.14 by a blue circle) in the common bridge between the base code and the parallelism modules (*i.e.*, concrete layer). This process corresponds to specifying the desired code transformations (including their customization) along with the base code join points where they should be applied. The cardinality between the code transformations and the base code join points is many-to-many. Thus, the same code transformation can be applied to several join points (*e.g.*, *pointcut critical_method() : call(public void Particles.*(...))* and Figure 5.9), and several code transformations can be applied to the same join point (*e.g.*, applying a DM and SM *for* distribution in the same *for* method). Such characteristics help to deal with the PRC tangling and scattering problems.

Composability

With AOP we built libraries composed of self-contained, and specialized, modules that tackle a specific PRC. Furthermore, these library modules are reusable and can be composed with a base code which is unaware of their presence. Consequently, these modules can be combined, and composed surreptitiously with the base code to provide SM, DM, and hybrid parallelizations.

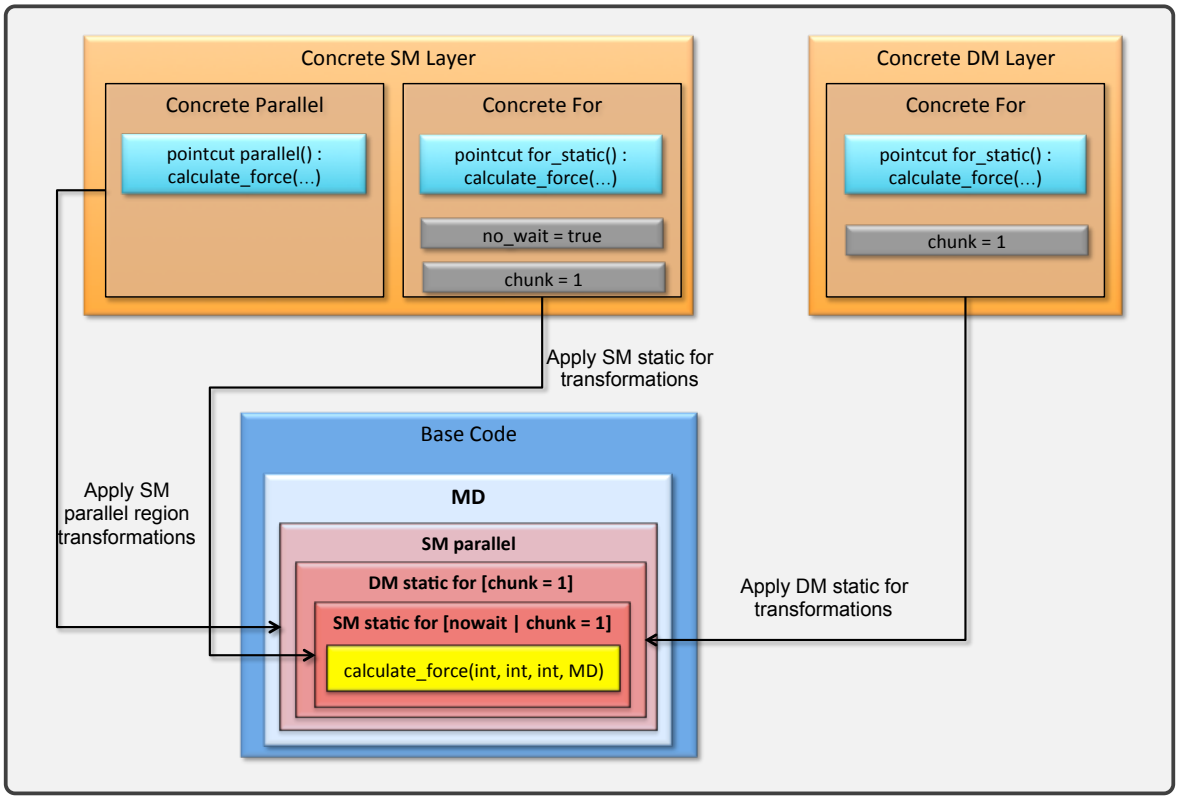


Figure 5.15: Example of the composition of the parallelism-related code transformations with the base code.

Figure 5.15 illustrates the composition of some of the framework parallelism-related constructors with the *MD* base class (part of the MD parallelization previously shown in Section 5.3.1). We will use this figure to explain some of the particularities of the composition process of our framework. The yellow, reddish and gray rectangles represent the extension point used to inject the parallelism-related code transformations (*i.e.*, the *calculate_force* method), the code transformations applied to the base code (*i.e.*, parallel region and the SM and DM *for* distributions) and the customization of some of these code transformations options (*i.e.*, chunk size and *nowait* options), respectively. Finally, for the sake of brevity let us label the process of composing parallelism-related code transformation(s) with an extension point as the *compositional step*.

With our approach, the user explicitly composes (inside the concrete layer) the parallelism-related code transformations with the base code by selecting them from the framework and specifying the extension points onto which they will be applied. As previously mentioned, this process can (always) be conducted by using inner aspects with pointcuts (*e.g.*, Figure 5.7) or (in some cases) using annotations with ITDs (*e.g.*, Figure 5.11). Technically, all code transformations provided by the framework can be combined. However, from a correctness point of view, some of these combinations are either undesirable or should be applied in a predefined order. For instance, applying a *master* around a *barrier* constructor would lead to deadlocks, and the majority of the SM constructors should be used in the scope of a parallel region.

The main difference in the *compositional step* using our framework in comparison to an intrusive approach, is that in the latter, the programmer manually, and intrusively defines the order (*i.e.*, before, after or around) in which the code transformations will be applied in relation to the extension point (and to each other). For instance, in an intrusive version of the example of Figure 5.15, the programmer would create a parallel region around the code that divides the outer loop iterations of the *calculate_force* method among processes and threads. However, in our approach the order in which the code transformations are applied is, by default, handled internally by the framework through the AspectJ mechanism to define the precedence order (discussed in Section 3.3.5). Thus, the framework user should be aware of this order, especially when applying different code transformations to the same extension point.

To understand the order in which the code transformations are applied during the *compositional step* of our framework, let us reason about the following two questions:

1. *In which order is a given code transformation injected into an extension point?*
2. *When there is more than one code transformation injected into the same extension point, in which order will they be applied?*

Regarding the first question, the default order in which the framework code transformations are applied is very intuitive. These code transformations can be injected simultaneously before and after (*i.e.*, around), or either only before or after an extension point. We label as *around transformations* the code transformations that are injected around an extension point (*e.g.*, *master*) and the remaining ones (*e.g.*, *barrier*) as *single point transformations*. Most single point transformations are applied by default after an extension point – except obvious ones such as *inject_code_before* that injects behavior before a method (*e.g.*, Figure 5.12). Nevertheless, for each constructor that performs single point transformations, the framework provides three versions of them, namely those that inject the code transformations before, after or with the constructor default value (which can be either before or after, depending on the constructor). These versions are easily distinguishable from each other; the ones applied either before or after have the keyword *before* or *after* on their names, respectively, whereas the default version does

not. Hence, as with the intrusive approach, with our framework, the user can explicitly dictate the order in which the single point transformations are injected. Finally, figuring it out whether a constructor applies an around or a single point transformation is also straightforward. The user only needs to reason about what the constructor does. AOmpLib constructors mainly inject around code transformations, whereas AOdmLib communication constructors mostly inject single point code transformations.

Regarding the second question, the framework has lists of predefined rules of precedence to determine the order in which multiple code transformations are injected into the same extension point. Those lists (defined using the AspectJ's aspect precedence feature) establish the order of the code transformations of constructors belonging to the same library and between constructors of different libraries (*i.e.*, between the constructors of AOmpLib and AOdmLib, useful for hybrid parallelizations). In the example of Figure 5.15, three constructors injected behavior around the same extension point (*e.g.*, *calculate_force* method); based on the framework precedence rules the parallel region has the highest precedence of these three constructors followed by the DM and SM *for* distributions. When the SM and DM *for* distributions are simultaneously injected into the same extension point (*i.e.*, *for* method), the loop iterations of the *for* method will first be divided among processes, and then among the threads of each process. However, this does not imply that the order of transformation is always DM constructors first and SM constructors second. Nevertheless, the order in which the code transformations of constructors applied to the same extension point are injected is also fairly intuitive (*e.g.*, in the AOmpLib the parallel region is the constructor with the highest precedence). Furthermore, the framework precedence rules only affect sound combinations of code transformations; for instance, it does not try to solve undesirable combinations such as a *single* over a *master* and *vice-versa*.

For the most part, it is expectable that the framework constructors will be used in a particular natural order (*e.g.*, a parallel region and then a *master* and not the other way around). Those assumptions facilitated the specifications of the precedence rules used by default. It is worth emphasizing that these precedence rules only affect the order of code transformations that are simultaneously applied to the same extension point. Therefore, the user only has to consider them in highly specific scenarios. Nonetheless, the framework provides SM and DM layers without predefined precedence rules for the cases in which it is undesirable to use the default precedence rules. In such cases, the user establishes the order of the code transformations of the *conflicting* constructors by defining in the (concrete) layer, the precedence rules of the aspects where these constructors belong (*e.g.*, *declare precedence : ..., ...*). Such an approach can also be useful when the user creates his/her (own) constructors since it allows the specification of the order that the code transformations of the user-defined constructors will be injected when in conflict with the other framework constructors.

Usability

The users of our framework have access to libraries based on commonly used parallelism-related models (*i.e.*, multithreading and message-passing) and influenced by well-establish standards (*i.e.*, OpenMP and MPI). Furthermore, besides the constructors from these standards, users can also access higher-level constructors, such as dynamic schedulers, reduction of multidimensional arrays and so on, which saves them from some of the low-level details of PP. All such low-level details are hidden behind high-level code transformations that can be invoked through pointcuts and annotations. By providing code transformations based on familiar models and standards, the framework reduces the parallelism-related concepts that a PP developer must learn.

```
1  public class A {
2    public void f(){ ... }
3  }
4
5  public class A_Extension extends A{
6    @Override public void f(){
7      // code transformations
8        super.f();
9      // code transformations
10   }
11 }
```

Figure 5.16: Use of class extension and method overriding OO features to extend the functionality of the base code.

From the perspective of PP developer, the set of new concepts that must be learned to use our framework comes mainly from how the parallelism is injected into the base code (*i.e.*, the concept of parallelism layers). Nonetheless, these concepts are familiar to OO programmers; concepts analogous to extending the functionality of an application through the use of class extension and method overriding OO features. For instance, in the example of Figure 5.16, the programmer extended the functionality of the *f* method by overriding it in the subclass *A_Extension*. Conceptually, our code transformations extend the functionality of the base code similarly to the example of Figure 5.16 – analogously, our code transformations do the same as the subclass of *A* (*i.e.*, *A_Extension*). The steps of choosing which method will be overridden and implementing the extra functionality (lines 7 and 9) are in our approach analogous to choosing the extension points (*e.g.*, *call(...forceUpdate(...))*) and code transformations (*e.g.*, *pointcut critical_method*), respectively. Moreover, the customization of the parallelism-related constructors' options (*e.g.*, *chunk = 1*) can be thought of as the overriding of parts of the default implementation of a method that uses the *template method* pattern [GHJV95] – a pattern used also in some of our *advices*. Finally, our design rules follow the same reasoning as extracting a method out of a block of code so that it can be overridden in the future. One difference, though, is that our framework can also extend the functionality of extension points other than methods, namely object fields. In conclusion, the familiarity of our approach with well-known OO concepts used to extend functionality makes it easier for OO developers to grasp our approach.

Even though our parallelism layers can add functionality to fields, their expectable interface is very restrained (*i.e.*, merely method calls and field accesses). A limited interface helps to control the complexity and increase the stability of the parallel applications developed with our approach and makes it easier for programmers to reason about the interactions between base code and parallelism layers (*i.e.*, can only happen through well-defined and stable extension points). To make those advantages even more noticeable, we propose that the code transformations from our framework should only be applied to public or protected methods that are neither final nor static. In Java private, final, and static methods cannot be overridden; hence, our approach should follow the same reasoning. Consequently, by using such a methodology, we reduce even more the conceptual gap between extending the base code functionality through class extension and method overriding and using our framework. Thus, providing an approach that enables the users of our framework to add parallelism to domain code, using familiar concepts analogous to class extension and method overriding but without the limitations of OO inheritance [Kuc04]. A programmer reasoning about a base code parallelized with our framework will follow a similar thought process that an OO programmer would about a base code where the parallelism was added through subclassing. Finally, in our approach, the use of the AspectJ *privileged* feature is discouraged; instead, layers that directly access private variables of the base classes should do it using proper setters and getters (*e.g.*, lines 14, 16 and 24 of the DM parallelization of the MD case study shown in Figure 5.12). These guidelines facilitate the development of parallelizations with our framework, making it less complex, more structured and maintainable.

We used AspectJ to materialize the concept of parallelism layers, and consequently, it is a technology that the user must understand to use our framework correctly. Nonetheless, for the development of parallel applications, our approach relies on a small subset of AspectJ pointcuts. To avoid confusions about the particularities between *call* and *execution* pointcuts (described in Section 3.3.2.1), for method calls our layers solely rely on the *call* pointcut, and never on the *execution* pointcut. Additionally, the programmer needs the *set/get* pointcuts for the fields, *within* and *withincode* to limit the scope of the extension points to be intercepted and the pointcuts *args*, *this*, and *target* to expose context. Notwithstanding, a significant number of the framework constructors can be used through annotations and ITD, which offers a more concise and intuitive syntax than pointcuts. To reduce the need to create *advices* our framework provides modules to support additional code transformations, such as injecting behavior before and after a method call, manipulating arguments of a method, and creating user-defined loop distributions, which would typically be implemented with a user-defined *advice*.

Extensibility and reusability

The framework provides a set of common code transformations that can be reused in different contexts. Nonetheless, our framework still has room for customization. The programmer can customize and extend existing modules or build his/her modules and incorporate them into our framework. These modules are then ready to be reused in future parallelism layers, thus building a repertoire of parallelism-related constructors. The customization and extension of the parallelism-related modules are also familiar to OO programmers (*i.e.*, performed through method overriding and a mechanism analogous to extending a class, respectively). Naturally to extend our framework programmers must be knowledgeable about the AspectJ technology (*e.g.*, creation of *advices*). Furthermore, programmers must be able to define the precedences of their modules, and those precedences must be in harmony with the other framework code transformations. Alternatively, the newly created user-defined code transformations can be left without precedences, and it is up to the framework user to define them, if so needed, when developing the concrete parallelism layer. Finally, programmers should also have an overview of how the framework is internally structured.

For each library, there are interfaces that define the *contract* between the aspect side and external services side, which the layer depends on (labeled "External SM APIs" and "External DM APIs" in Figure 5.13). Those interfaces define what the libraries of aspects expect from the external API and *vice-versa*. The low-level details about threads/processes pools and so on are hidden behind the external API side, leaving aspects with the logic of requesting actions from the external API to meet the parallelism expressed in the layer. For example, when a parallel region is encountered, the aspect side sends to the external API a task and the number of workers that should execute that task in parallel. In this case, the task is the execution of a block of code (*i.e.*, a method from the base code) with a barrier at the end of it (if the user did specify otherwise). The aspect side is unaware of the type of barrier and pool that will be used to perform that task. This separation of concerns makes the framework more adaptable to future changes and allows programmers to interchange between different external APIs without having to change the core of abstract layers, as long as the interfaces are respected. Nonetheless, the programmer can still directly access the low-level details such as thread/process ID and so forth through calls from the aspect side to the external API side. Consequently, such low-level details can also be accessed from the concrete layer as well, which can be helpful during the parallelism tuning process or the creation of highly specific user-defined code transformations.

The framework is implemented by multiple abstract aspects, among other reasons, because it helps to break down the internal complexity of the libraries. The same reasoning was also applied to the pointcuts as well. The elementary pointcut [BLJT07, LJ06] idiom was applied to some of the abstract pointcuts to divide them into groups of sub-pointcuts to represent fix and variable state. The latter is typically related with context, for instance, the argument of

a method. The fix and the variant parts of the pointcuts are respectively final and non-final. Hence, the programmer is allowed to override the latter one. All the pointcuts that can be reused by parallelism modules provided by the framework are stored in a proper aspect higher in the hierarchy of aspects. Most of the times, those pointcuts are only useful internally. Nevertheless, they can be used in the concrete layer aspect. Thus, the programmer can use them to restrict the scope of other pointcuts declared in the concrete layer.

Efficiency

Regarding efficiency, our framework performs most of their code transformations at compile-time, which avoids unnecessary overheads. Special care was taken to avoid as much as possible the use of AspectJ mechanisms that are likely to introduce high overheads (*e.g.*, *cflow* and reflection API). The SM and DM constructors provided by the framework are based on Java concurrency framework and highly efficient MPI implementations, respectively. From the perspective of the framework user, the fact that the user can extend and create their own parallelism-related code transformations and modules, and that it is possible to access the low-level details of PRC from the concrete layer (*e.g.*, thread/process ID) allows for the creation of highly-tuned, and possible domain specific, code transformations. Consequently, enabling the development of efficient parallelizations.

5.3.3 Design rules

Computational and data related transformations

The framework provides parallelism-related constructors that might require two types of code designing, namely *method refactoring* and, sporadically, turning local variables of a primitive data type into objects or fields (*i.e.*, *extract field refactoring*). The majority of the constructors provided by the framework are applied to methods. Thus, the points in the base code on which those constructors will be injected have to be methods (*i.e.*, *method* design rule). Additionally, context (*e.g.*, loop range) may have to be passed to constructors using the arguments, or returning type, of those methods. Dependencies inside those methods that can lead to different results when executing in parallel must be resolved by the programmer (*e.g.*, dependencies between iterations of a parallel loop). Furthermore, to prevent undefined behaviors, the methods in which the PRC are injected should:

- return void, unless when explicitly allowed or required by the constructors;
- contain just the relevant code for the parallelism-related constructor. For example, the *for* method should encapsulate only the *loop* (along with its body) to be parallelized.

To deduce if, and how many, arguments are needed to pass context to a particular parallelism-related constructor, the programmer merely needs to inspect the pointcut signature of that constructor. For instance, a pointcut with the signature *someConstructor(int)* expects that one argument is used to pass context.

Regarding the position of the arguments that are used to pass context to the parallelism-related constructors, the framework only constraints the *for method*. The *for* loop to be parallelized must be encapsulated into a method, and its range passed as the first three parameters of that method. For the collapsed loops the first `3 * N` positions should be used, with `N` being the number of nested loops to collapse. Finally, for loops iterating over a collection that implements the Java interface *java.util.List*, only one parameter (using the first position) needs to be passed, the list itself, since the range is implicitly deduced. These constraints exist because the framework provides annotations for parallel loops. With annotations, in contrast to pointcuts (*i.e.*, *args(...)*), there is no reasonable way to specify the positions of the arguments that should be *read* by the aspects. With these constraints, the framework can assume the position of the arguments that correspond to the loop range and, consequently, enable the use of annotations to specify *for* constructors. The framework accepts the following three loop forms:

- `for(int i = begin; i < end; i += c)` with `end > begin`;
- `for(int i = begin; i > end; i -= c)` with `begin > end`;
- `for(E e : list)` with : `list implements List < E >`.

The loops of the form `for`(...;...;...) that are allowed by OpenMP can be turned into one of the first two. By relying on methods to inject parallelism, the points in the base code where the PRC is added can be uniquely identified, and additional behavior can be easily (un)plugged. Conceptually, this rule complies with OO philosophy; extending the methods of the base codes with PRC follows the principle of method overriding.

Constructors that apply data-related transformations (DRT) are applied to objects or fields. Hence, forcing primitive local variables to be turned into either objects or fields. However, in practice, this scenario only happens when applying AOdmLib DRT to primitive local variables.

All the design rules were carefully thought out to guarantee the sequential semantics of the base code. Hence, if all the layers are unplugged, the base code will still work correctly. Nowadays, most IDEs can automatically perform the code refactors needed by our design rules. From now on the acronym IDE, will be used to refer to the Eclipse IDE. Section 5.4.1 explains in greater detail the reasoning behind all the design rules previously mentioned.

Performance design rules

This small set of extra design rules is not mandatory to be applied but are useful to increase performance. The first performance design rule is for code transformations related to the interception of object creation. According to this rule, the user can turn the object creation into a method that returns the reference to the object to be created. Some practical cases where this design rule can be applied are: - the creation of a set of locks to lock accesses to the elements of a given array; - the user wants to split an array across multiple processes. Figures 5.17 and 5.18 show a code before and after applying the *object creation* design rule, respectively.

```
1 void method(...){
2   A = new double [N][M]; ...
3 }
```

Figure 5.17: Before applying the *object creation* design rule.

```
1 double [][] newMatrixA(int N, int M) { return new double [N][M];}
2
3 void method(...){
4   A = newMatrixA(N, M);
5   ...
6 }
```

Figure 5.18: After applying the *object creation* design rule.

To understand the performance implication of not applying this design rule, let us consider that the matrix A of Figures 5.17 and 5.18 was a field and that the user wanted to split that matrix by lines across `N` processes. Hence, each process should allocate only a matrix of size `1xM`. AspectJ's *set* pointcut could have been used to intercept its creation, and internally

the framework would ensure that each process only allocates its assigned matrix chunk. The problem is that AspectJ will only intercept field A after the return of the reference by the *new* constructor. Thus, processes will first allocate a temporary matrix of `NxM`, reallocate the matrix with the actual correct size and assign it to the field A. This leads to unnecessary memory allocation[9]. However, by using a method that returns the reference, the framework can first intercept the call of the method, ensure that only the master thread/process allocates the entire matrix and then instruct the remaining threads/processes.

```
1 void code(){
2
3   for(i = 0; i < 100; i++)
4     x = x + i;
5
6 }
```

Figure 5.19: Before applying the *set/get* performance design rule.

```
1 void code(){
2   int x = this.x;
3   for(i = 0; i < 100; i++)
4     x = x + i;
5   this.x = x;
6 }
```

Figure 5.20: After applying the *set/get* performance design rule.

Our second performance design rule is related to the use of *set/get* DRT pointcuts, which are used by the framework to intercept fields and apply additional logic into them. For instance, to make the field x of Figure 5.19 thread private, the SM library intercepts the accesses to that field (line 4) and replace it for accesses to the correspondent thread copy of that field. From a performance point of view, the problem of this approach is that for every access to the field to be intercepted, the library adds additional instructions. These extra instructions might add a significant overhead, especially inside loops. Therefore, our *set/get performance* design rule (Figure 5.20) focus on reducing the number of interceptions done by the framework. To achieve that, the user creates a local variable that temporally replaces the use of the field within the *DRT method*[10]. Moreover, the local variable needs to be initialized with the value of the field and broadcast back the final result to the field, both in the begin and at the end of the code of the *DRT method*. The lines 3 and 6 of Figure 5.20 show the introduction of those anchor points. Using the same name of the field for the local variable automatically replaces any access to the field x, in that method, by accesses to the local variable x. By applying the design rule in the code of Figure 5.19, it was possible to reduce the number of interceptions from 200 to only 2. In practice, the framework only needs the point from line 3. However, the assignment in line 5 is important (for primitive variables) to keep the sequential semantics of the base code.

In intrusive parallelizations, similar performance design rules are frequently applied. For instance, Java thread-local class is commonly used to provide threads with a private copy of an object. A similar optimization to the one shown in Figure 5.19 can also be used to reduce the number of calls to the get/set methods of that class.

[9] In some of our tests we even had *out of memory* problems.
[10] The method where the data-related transformations are injected into.

5.3.4 Shared Memory Library

5.3.4.1 Execution model and computational transformations

OpenMP annotations (*i.e.*, *directives*) signal the points in the base code where PRC should be inserted. The C/C++ OpenMP standard requires that some directives are applied on top of a single statement, which often results in code statements being enclosed in brackets to form a single logical block. The constructors of AOmpLib are injected into method calls and fields.

As with OpenMP, the AOmpLib execution model is based on the fork-join model [apiva] with a focus mainly on loop and task parallelism. In AOmpLib, programs start with a single thread named *initial thread* (using the OpenMP nomenclature). Since all examples presented in this work do not use nested parallelism, we labeled the *initial thread* as master thread or just master. When the master reaches a method that is intercepted by the parallel region pointcut *parallel* (or the annotation *@Parallel*) a new team of threads is created. All the threads from that new team execute, in parallel, the intercepted method and wait for each other at the end of it (unless the user explicitly requests otherwise). When a team of threads encounters a work-sharing constructor (*e.g.*, *for* distributions, *sections* and *single*) work is assigned to them.

```
1  static aspect parallel_Region extends Sm_Parallel
2  {
3    pointcut parallel() : call (/** some method **/)
4                          && if(/** some conditional **/);
5
6    @Override
7    public int num_threads(){
8      return ...;
9    }
10
11   @Override
12   public boolean no_wait(){
13     return ...;
14   }
15 }
```

Figure 5.21: AOmpLib - Parallel region pointcut example.

Figures 5.21 and 5.22 illustrate a parallel region using pointcuts and annotations, respectively. With AOmpLib the user can specify, for instance, the number of threads that will execute the parallel region, conditions that have to be met for that parallel region to be created (line 4 of Figure 5.21), among others.

```
1 declare @method /** some method **/: @Parallel(num_threads=..., no_wait=...);
```

Figure 5.22: AOmpLib - Parallel region annotation example.

AOmpLib also provides the OpenMP *static*, *dynamic*, *guided*, and *runtime* loop distributions, but does not provide *auto*. By choosing *auto*, the user delegates to the compiler the decision of selecting the most suitable distribution. Since our framework does not perform analysis of loops, it cannot provide the *auto* feature. For each of the supported loop distributions, the AOmpLib has a specific pointcut (*e.g.*, *for_dynamic*) and annotation (*e.g.*, *@For_dynamic*). AOmpLib supports the *collapse* option for two nested loops and, since AspectJ does not support pointcut overloading, AOmpLib provides unique pointcuts and annotations for the collapsed loops as well (*e.g.*, *for_dynamic2*). Moreover, AOmpLib provides the *ordered* constructor to ensure that a section of code inside a parallel loop is executed in the same order as it would if the loop was executed sequentially.

The constructor *sections* (shown in Figure 5.23) allows the assignment of heterogeneous workloads to different threads within a team. This constructor should be used inside a parallel region so that the multiple sections can be executed in parallel. For this constructor, the user needs to identify the code sections (*i.e.*, method calls) that will be executed asynchronously, and the method that calls these sections (line 3 of Figure 5.23).

```
1 static aspect sections extends Sm_Sections
2 {
3   pointcut sections() : call (/** some method **/);
4
5   pointcut section() : call (/** method 1 **/) ||
6                        call (/** method 2 **/) ||
7                        call (/** method 3 **/);
8 }
```

Figure 5.23: AOmpLib - Sections pointcut example.

Constructors such as parallel region, *for* and *sections* have implicit synchronization barriers at the end of their execution. Nonetheless, users can explicitly request additional synchronization points among threads in a team. For instance, AOmpLib provides OpenMP alike constructors such as *critical* and *barrier*. Methods declared as *critical* are executed sequentially, and barriers function as single points of synchronization that force threads (of a team) to wait for each other. AOmpLib offers two variants of the *barrier* constructor, namely *barrier_before* and *barrier_after* (or simply *barrier*), that introduce barriers before and after a method call, respectively. Additionally, AOmpLib offers the *critical_method* constructor, which works as a *synchronized* method in Java (*i.e.*, locks using the object where the method belongs to). As with OpenMP, AOmpLib also offers the use of explicit locks – including arrays of locks. However, unlike OpenMP, with AOmpLib those locks can be used without having to be explicitly called in the base code. AOmpLib also offers specialized read and write locks as well as atomic operations. The latter is provided by the classes (*e.g.*, *AtomicInteger* and *DoubleAccumulator*) from the Java package *java.util.concurrent.atomic* [Cla]. Most of the remaining AOmpLib constructors (*e.g.*, *single*) also follow the OpenMP standard.

5.3.4.2 Data model and data-related transformations

In Java, there are instance variables that can be static (*i.e.*, class variables) or non-static (*i.e.*, object fields), local variables and parameter variables. In the context of AOmpLib data model, it is also convenient to divide these variables into objects and primitive data types, variables with and without an explicit accessible memory reference, respectively.

Static variables are shared among threads, and objects created before a parallel region are shared by the threads inside that region. Local variables created inside a parallel region are private to each thread. Finally, method arguments are passed by value; when primitive variables or objects are used as a parameter, a new copy of their value/reference is created and used as the actual method parameter. Hence, arguments are also private to threads[11].

AOmpLib follows the Java memory model, by default. In AOmpLib, from the threads point of view, data can be shared or private. With AOmpLib the user can apply DRT such as *private*, *firstprivate*, *lastprivate* and *reduction*. AOmpLib can reduce arrays, matrices and also other types of objects when the user-defined reductions are defined. As in OpenMP, in AOmpLib it is possible to make an object private to threads as long as the copy class constructor is provided. However, with AOmpLib the user can further specify different DRT to be applied to the fields of these thread private objects. For example, the user can provide a copy class constructor that merely performs a shadow copy of an object. In this case, each thread has a private copy of an object with fields of primitive data type that are private, while the remaining are shared (*i.e.*, reference type). With this in mind, the user can then use AOmpLib annotations or pointcuts to apply DRT at the field level. Nevertheless, there are some restrictions to this workflow:

1. if fields within the private object are themselves objects, DRT can only be used on them if the AOmpLib supports them (*e.g.*, arrays);

2. annotations can only be used in the *direct* fields of the object to become private (*i.e.*, annotations cannot be used in fields of another field and so forth).

In the first case, the AOmpLib offers DRT for primitive variables and a set of object types. For the remaining objects, the library provides an appropriate interface that can be implemented by them. This interface will request the user to implement methods as *copy(...)*, *reduction(...)*, among other, making those objects supported by the AOmpLib as well. Finally, the last case relates to the reduction of complexity and overhead of the AOmpLib implementation.

[11] When a method argument is an object, its reference is private to threads but threads work with the same memory location.

To apply the DRT, the AOmpLib provides three approaches, namely *method pointcuts*, *set/get pointcuts* and the *private object pointcut and annotations*. The first and second approaches are used to apply DRT to local objects and fields, respectively. The third is to create private objects *per* thread and apply DRT into their fields.

With the *method pointcuts* approach, the user specifies a pointcut with the method call to intercept (*i.e.*, *DRT method*). Since the data will be passed as a parameter of the *DRT method*, the *args* pointcut should be used to specify the argument position (line 2 of Figure 5.24).

```
1  pointcut reduction_arg(double [][] mA) : call( /** Some Method **/)
2                                           && args(double[][], ...)
```

Figure 5.24: AOmpLib - *Method pointcuts* example.

In the *set/get pointcuts* approach, the user specifies the method associated with the DRT (line 1 of Figure 5.25) and additionally the three *w's*, namely *which*, *when* and *what*. These refer to *which* data will be intercepted, *when* it will be intercepted and *what* DRT should be applied to that data. Figure 5.25 shows the definition of pointcuts to apply the DRT using the *set/get pointcuts* approach. The lines 2 and 3 of Figure 5.25 represent user-defined pointcuts[12] whereas the lines 1 and 4 are AOmpLib pointcuts. Line 2 declares *which* data will be used in the DRT, while line 3 specifies *when* the data from line 2 should be intercepted. Finally, line 4 defines *what* should be done with that data, in this case, a reduction with the sum operator. As shown in Figure 5.25, this approach requires the use of *set/get* pointcuts to intercept the writes/reads to/from the desired fields, respectively. The *withincode* pointcut was also needed to restrict the scope of the variable to be intercepted.

```
1  pointcut private_data() : call(DRT_method(..));
2  pointcut data()         : get(* *.MD.epot) || set(* *.MD.epot);
3  pointcut dataScope()    : withincode(... void addEpot (double));
4  pointcut reductionSUM() : data() && dataScope();
```

Figure 5.25: AOmpLib - *Set/Get pointcut* DRT example.

The *private object pointcut and annotations* approach is used to create private objects and apply DRT to their fields. In this use case, as with the previous one, it is necessary to declare the *which*, *when* and *what*. However, the *which* includes specifying the object and possibly its fields. Additionally, as with OpenMP, the user must also provide the copy constructor of the object to become private (line 6 of Figure 5.26). As illustrated in line 1 of Figure 5.26, annotations are used to specify both the *which* fields and *what* to do with them, whereas the pointcuts in lines 4 and 5 are used to specify the *when* and *which* object, respectively. The object to become private is always related to the *DRT method*. The pointcuts to express the object to become private can be *this*, *target* or *args*.

[12] These pointcuts are optional; they could have been used directly used in line 4.

```
1 @SmData (List = {@OP(Type = Type.SUM, vars = {"epot"})})
2 static aspect privateData extends Sm_PrivateObject <T>
3 {
4   pointcut private_object(T object) : call(... DRT_method(..))
5                                       && target(object);
6   public T T.copy() {
7     ...
8     return ...;
9   }
10 }
```

Figure 5.26: AOmpLib - Private object pointcut and annotations example.

In the *private object pointcut and annotations* approach, the use of annotations to specify the fields offers a cleaner solution than using the *set/get pointcuts* approach. Worth noting that if the fields specified in the annotations do not exist AOmpLib throws an exception.

	(First) Private	Last Private/Reduction
Shared local object	method pointcut	
Field of a shared object	set/get pointcuts	
Shared object and its fields	private object pointcuts/annotations	

Table 5.1: AOmpLib - The summary of data-related transformations.

Although several pointcuts to apply DRT were exposed, Table 5.1 shows that it is not hard to reasoning about them. Because AOmpLib offers the possibility of adding DRT to objects and its fields independently, our DRT use cases seem more complicated that OpenMP 3.1. Our DRT use cases would have been much simpler if the AOmpLib only allowed the application of DRT at the field level. However, extending these transformations to local objects provides more flexibility. Otherwise, local arrays, for example, would have to be transformed into an object field. Nevertheless, regardless if it is OpenMP 3.1 or AOmpLib, the programmer still has to reason about the scope of the variables in a SM parallelization, especially because of potential race conditions. Part of this reasoning is initially done when applying the method refactoring. As we can see in Table 5.1, and as it would be expected, the private-related DRT of AOmpLib are only applied to variables that are shared among threads.

5.3.5 Distributed Memory Library

Besides the SM library (*i.e.*, AOmpLib), our framework also provides a DM library (*i.e.*, AOdmLib) based on the MPI standard, with a set of modules that can be composed with a base code to support its execution in DM environments (*e.g.*, clusters). AOdmLib is built on top of OpenMPI [VGRGS13], which besides being one of the most popular implementations of MPI [VGRS16], also supports Java [VGRGS13] using a JNI-based approach.

The execution model of AOdmLib is similar to the SPMD of MPI. The AOdmLib starts by intercepting the execution of the *main* method of a given project (specified by the user), initializes the MPI processes with the option *MPI_THREAD_FUNNELED*[13], creates all the necessary data structures and then executes the *main* method. Additionally, AOdmLib ensures that after the execution of the *main* method the MPI *finalize* function will be called. During the execution of the *main* method, OpenMPI ensures that there are as many instances of the same base program running in parallel as the number of processes specified by the user. Each MPI process executes a JVM instance with a copy of the base program and its data.

The AOdmLib is built on top of the MPI API, and consequently offers most of the MPI constructors, namely *Allreduce*, *gather*, *scatter*, *broadcast* and many others. Additionally, the AOdmLib offers also constructors that are not covered by the MPI standard, such as loop distributions, several strategies to split/collect multidimensional arrays among/from processes (*e.g.*, dividing matrices by columns and lines), the possibility of transparently use in the MPI communication routines other objects than just arrays (*e.g.*, vectors), and restricting methods to be executed by only a certain process.

As with AOmpLib, AOdmLib provides the *for* constructor with the possibility of choosing different distributions, namely dynamic, static (*e.g.*, Figure 5.12), and user-defined (*e.g.*, Figure 5.27). From the framework user point of view, invoking the *for* constructor from the AOmpLib is the same as invoking it from the AOdmLib – as it can be confirmed by comparing the SM and DM *for_static* constructors previously shown in Figures 5.7 and 5.12, respectively. Furthermore, both libraries also provide the same annotations to invoke the *for* constructors (*e.g.*, *@For_static(chunk=1)*).

Figure 5.27 shows an example of the user-defined *for* distribution constructor of the AOdmLib. This constructor works as an AspectJ *advice*. However, instead of using an actual *advice*, the user overrides the *for_user_defined* method (line 5 of Figure 5.27) – that exposes the loop range – and calls the *for_method* (lines 7 and 8) to call the original method (*i.e.*, the *for* method of the base code). For obvious reasons, neither AOdmLib nor AOmpLib provides annotations for this constructor.

[13] *"The process may be multi-threaded, but only the main thread will make MPI calls (all MPI calls are funneled to the main thread)"*[tMee]

```
1 static aspect forceCalculation extends Dm_For {
2     pointcut for_user_defined() : ....
3
4   @Override
5   public void for_user_defined(..., int begin, int end, int step) {
6     ...
7     for_method(getProcessID(), halfEnd, getTotalProcesses());
8     for_method(halfEnd+process1Iteration2half, end, getTotalProcesses());
9   }
10 }
```

Figure 5.27: AOdmLib - User-defined *for* pointcut.

In OpenMPI, and consequently in AOdmLib as well, each MPI process has a unique ID that ranges from `0` to `N`, with `N` being the total number of processes chosen by the user. The process with ID zero is named the *master* process, or simply *master*. In the example of Figure 5.27, the user could make his/her customized loop because AOdmLib provides methods to get the total number of processes running the application in parallel and the ID of the current process.

In the DM parallelizations, partitioning data among processes is an approach used more often than dividing the iterations of a loop since it reduces the amount of data to be exchanged and the memory footprint *per* process. For instance, typically in matrix multiplications (example detailed in Section 6.3), the SM parallelization will distribute the iterations of the loops of its main kernel among threads, whereas the DM parallelization will scatter some of its matrices among processes. AOmpLib offers specialized modules that implement the partitioning among processes of commonly used data structures, such as uni- and multi-dimensional arrays, vectors, and others.

```
1 static aspect splitRowsMatrixA extends DM_Share <MM> {
2   ...
3   pointcut scatter_creation() : call(... double[][] newMatrixA(..));
4   pointcut global_view()      : call(...);
5
6   pointcut scatter_before()   : call(...);
7   pointcut gather_after()     : call(...);
8
9   pointcut get_global_index(...) : call(...)
10                                   && args(...);
11
12  @Override
13  public void data(MM baseClass) {
14    baseClass.A = create_view(baseClass.A);
15  }
16
17  @Override
18  public int chunk_lines(MM baseClass){
19    return baseClass.tilei;
20  }
21  ...
22 }
```

Figure 5.28: AOdmLib - Data partitioning specialized module.

Figure 5.28 shows an example of using the partitioning data module from AOmpLib to split a matrix (A) by chunks of lines with a certain size (lines 17 to 20 of Figure 5.28). To specify the data (*i.e.*, matrix A) – that will be associated with the current instance of the partitioning module (*i.e.*, *splitRowsMatrixA* defined in line 1) – and its creation, the user must define the *data* method (lines 12 to 15) and the pointcut *scatter_creation* (line 3), respectively. After intercepting the extension point associated with the pointcut *scatter_creation*, the matrix A will be scattered (by the master) across processes according to the options chosen by the user. In this example, the option chosen was to scatter the matrix by chunks of lines with a size of *tilei* (lines 17 to 20). Besides the chunk, it is also possible to define other options, such as splitting by columns, creating *ghost cells*, among others.

In reality, with the *scatter_creation* constructor, the master will not send to each process the data to be scattered, instead will send a message with the dimensions of the sub-matrix that each of them should allocate. We use this strategy since there are cases in which the processes are only interested in the size of their sub-matrix and not concern about the content of the original matrix. Hence, in such cases, our approach saves unnecessary communication. To actually scatter the original matrix among processes, the user can use the *scatter_before* (*e.g.*, line 6 of Figure 5.28) and *scatter_after* constructors that perform the scattering before and after an extension point, respectively. The *gather_before* and *gather_after* (*e.g.*, line 7) constructors allow to collect the content of the sub-matrices and merged it back into the original matrix – allocated in the master. For matrices that will be scattered/gathered by lines, AOdmLib also gives the option that during the data allocation each process allocates a complete matrix with some of its lines set to null[14] instead of the sub-matrix. This option can be helpful in parallelizations in which it is desirable that all processes are aware of the dimensions of the original matrix. In the partitioning data module, with or without the *nulls* option, the master process has always the original matrix.

AOmpLib offers pointcuts and methods that receive a local data index from a process (*e.g.*, lines 9 and 10 of Figure 5.28) and return what would have been the correspondent position in the original data (*i.e.*, global index), and *vice-versa* (*i.e.*, *get_local_index*). Regardless of the data type being used in the partitioning module (*e.g.*, matrix, array, and vector), the constructors are the same, it is up to AOdmLib to internally figure it out the data type being used.

Using the options defined in Figure 5.28 for a matrix A with a size of `1024x1024` and *tilei* of 512, in an application running in parallel with two MPI processes, each of them would initialize a sub-matrix of size `512x1024`. During the scattering phase (*e.g.*, *scatter_before*) the process 0 (*i.e.*, master) would take from the original matrix A the content of the lines 0 to 511, whereas the process 1 would take the content of the lines 512 to 1023. Finally, for the process with ID `1`, the local index (0,0) would correspond to the position (512,0) in the original matrix.

[14]Naturally, the master will allocate the entire matrix without any line set to null.

Since the master holds a subset of the data (*i.e.*, *local view*) as well as the original data (*i.e.*, *global view*), AOdmLib allows that process to temporally switch between local and global views, during the execution of a method. During the execution of the methods intercepted by the constructor *global_view* (line 4 of Figure 5.28), the master works with the complete data. Worth noting that those methods are only executed by the master, the remaining processes will ignore these methods. Moreover, the master view over the global data might not reflect the latest changes made by the other processes. An up to date view over the global data is only possible with the *gather* operator. After a gather operation, the master process will "permanently" switch the view from local to global until a new scatter operation occurs.

```
1  pointcut scatter_before()    : call(...);
2  pointcut gather_after()      : call(...);
3
4  @Override
5  public void scatter(MM sourceClass) { /** user-defined scatter **/ }
6
7  @Override
8  public void gather(MM sourceClass){ /** user-defined gather **/ }
```

Figure 5.29: AOdmLib - Customization of the scatter and gather constructors.

For unconventional scatters and gathers, the user can override the AOdmLib scatter and gather default methods (lines 4 and 8 of Figure 5.29). To facilitate this customization, AOdmLib offers API to access, among others, the local and global data views.

AOdmLib also offers a module that replicates data, instead of splitting/collecting it across/from processes. Hence, for a given data structure assigned to a process, this module creates a copy of that data. From the user perspective, this module works similarly to the partitioning module, and it can be useful for parallelizations where it is desirable to have temporary buffers or save state in intermediate data structures to avoid polluting the original data.

```
1  static aspect Communication extends Dm_Comm <MM>{
2    pointcut comm_after() : call(... matrixMultiplication(...));
3
4    @Override
5    public void data (MM mm){
6      allReduce(mm.B , Reduction_DM_OP.SUM));
7    }
8  }
```

Figure 5.30: AOdmLib - Communication example.

Naturally, AOmpLib offers a module to specify communication routines such as *all reduction*, *reduction*, *broadcast*, and so on. For that, the user must specify when the communications should occur (line 2 of Figure 5.30) using the *comm_after* and *comm_before* constructors, and the communication to be performed inside the method *data* (lines 4 to 8). The module of Figure 5.30 ensures that all the communications defined inside the method *data* – *all reduction* in this example – will occur after the calls of the *matrixMultiplication* method.

5.4 Implementation

5.4.1 The reasoning behind the design rules

This section describes in detail the reasoning behind the design rules presented in Section 5.3.3.

5.4.1.1 Computational-related design rules

As previously described, the framework applies constraints to the argument positions of the *for methods*. A similar constraint was not applied to other constructors of the framework because it would increase the fragility of our approach. Moreover, it would be impossible to simultaneously use constructors that intercept the same join points if these constructors require different context to be passed through the same argument position.

Regarding the *method* design rule, there is an edge case worth mention. Let us imagine that we want to make a method out of the statements of lines 6 and 7 of Figure 5.31. The IDE will not be able to apply the method refactoring because there are primitive local variables updated inside the block of code to be refactored and their newly update values are used outside of that block of code (*i.e.*, variables a and b).

```
1  void code(){
2    // code block 1
3    int a = 10, b = 10;
4
5    // code block 2
6    a++;
7    b = a + 2 + b;
8
9    // code block 3
10   log("("+a+","+b+")");
11   a = b = 0;
12 }
```

Figure 5.31: Method refactoring problematic example.

```
1  void code(){
2    // code block 1
3    int a = 10, b = 10;
4
5    // code block 2
6    a++;
7    b = a + 2 + b;
8
9    // code block 3
10   a = b = 0;
11   log ("("+a+","+b+")");
12 }
```

Figure 5.32: Method refactoring clean example.

In the code of Figure 5.32, the IDE is robust enough to understand that the variable updates in line 6 and 7 will not matter in the third block of code, thus allowing to perform method refactoring on the second block. The described edge case seems unlikely to happen due to its specificity. Nevertheless, the IDE still warns[15] the programmer of such issues. To solve such scenarios, the programmer can try to agglomerate the two problematic blocks together. If that is not possible, the programmer can turn the conflicting variables into fields (*i.e.*, *extract field* refactoring) or apply the *introduce parameter object* refactoring [Obj]. These transformations can be performed automatically by the IDE.

[15] "*... Selected block modifies more than one local variable used in subsequence code. Affect variables are : ...*"

Our parallel loops have, for the most part, the same restrictions as those described in OpenMP 3.1 C/C++ standard[16]. However, there are some slight differences between our implementation and the one from OpenMP that are worth mention. Our framework accepts the following two canonical forms:

1. `for(int` $\jmath = \tau; \beta; \gamma$) with :
   a) $\tau \in \mathbb{Z}$;
   b) $\beta \in \{\jmath \circ \lambda, \lambda \circ \jmath\}$;
   c) $\circ \in \{ <, >\}$;
   d) $\lambda \in \mathbb{Z}$;
   e) $\gamma \in \{ \jmath$ `+= c,` $\jmath$ `-= c,` $\jmath$ `= `$\jmath$ `+ c,` $\jmath$ `= c + `$\jmath$ `,`$\jmath$ `= `$\jmath$ `- c`$\}$;
   f) `c` $\in \mathbb{N}^*$.
2. `for(E e : list)` with : `list implements List` $< E >$.

Some relevant restrictions imposed by the OpenMP standard on the first loop form are:

1. Informally, τ, λ and `c` represent the *for*' begin, end and incremental step, respectively. All of them have to stay constant throughout the loop execution. γ represents the incremental/decremental expressions – henceforth loop step – supported by our framework;
2. $\jmath$ is the variable that will hold the current loop iteration and can only be changed by the expressions in γ;
3. If $\beta \in \{\jmath < \lambda, \lambda > \jmath\}$ then $\gamma \in \{ \jmath$ `+= c,` $\jmath$ `= `$\jmath$ `+ c,` $\jmath$ `= c + `$\jmath\}$;
4. If $\beta \in \{\jmath > \lambda, \lambda < \jmath\}$ then $\gamma \in \{ \jmath$ `-= c,`$\jmath$ `= `$\jmath$ `- c`$\}$.

We decided to add two extra restrictions that are not explicitly defined by the OpenMP standard. For loops using $\beta \in \{\jmath < \lambda, \lambda > \jmath\}$ then $\tau < \lambda$ and for loops using $\beta \in \{\jmath > \lambda, \lambda < \jmath\}$ then $\tau > \lambda$. In other words, this means that when $\jmath$ increments between iterations the framework expects that the begin of the loop is smaller than its end. When the $\jmath$ decrements the framework expects that the begin of the loop is greater than its end. Those restrictions were added to make the *for method* design rule more readable (this will be clarified later).

For the first loop OpenMP allows four additional loop step expressions and two extra conditional operators, namely:

- $\gamma \in \{$`++`$\jmath$, $\jmath$`++`, `--`$\jmath$, $\jmath$`--`$\}$;
- $\circ \in \{ \leq, \geq\}$.

[16] From pages 40 to 46 of [apiva]

All those loop step expressions and conditional operators can be converted into the ones allowed by our framework. For instance, the additional loop step expressions can be reduced to actual two $\jmath$ `+= c` or $\jmath$ `-= c`. This transformation can be performed by applying the following equivalences:

- `++`$\jmath \equiv \jmath$`++` $\equiv \jmath$ `+= 1`;
- `--`$\jmath \equiv \jmath$`--` $\equiv \jmath$ `-= 1`.

It is also possible to transform both $\leq$ and $\geq$ into $<$ and $>$, respectively, by applying the following transformations:

- `for(int` $\jmath = \tau$; $\jmath$ `<=` λ; $\jmath$ `+= c)` $\equiv$ `for(int` $\jmath = \tau$; $\jmath <$ $\lambda +$ `c`; $\jmath$ `+= c)`
- `for(int` $\jmath = \tau$; $\jmath$ `>=` λ; $\jmath$ `-= c)` $\equiv$ `for(int` $\jmath = \tau$; $\jmath >$ $\lambda -$ `c`; $\jmath$ `-= c)`
- `for(int` $\jmath = \tau; \lambda >=$ $\jmath$; $\jmath$ `+= c)` $\equiv$ `for(int` $\jmath = \tau$; $\lambda +$ `c` $> \jmath$; $\jmath$ `+= c)`
- `for(int` $\jmath = \tau; \lambda <=$ $\jmath$; $\jmath$ `-= c)` $\equiv$ `for(int` $\jmath = \tau$; $\lambda -$ `c` $< \jmath$; $\jmath$ `-= c)`

Informally, this transformation is about adding one loop step on the loop's end (λ) and dropping the equals signal. If the loop follows the already described restrictions, to remove the $=$ from $\leq$ and $\geq$ we only need to look at the loop step and loop's end. Hence, if $\jmath$ `+= c` then the new loop end is $\lambda +$ `c`, likewise for $\jmath$ `-= c` then the new loop end is $\lambda -$ `c`. These rules might seem complicated, but they are intuitive especially because the most used loops are of the form `for(int i = begin; i < end; i++)`.

All the forms of the first type of loops that the framework and OpenMP support are semantically equivalent to one of the following two loops:

- `for(int i = begin; i < end; i += c)`;
- `for(int i = begin; i > end; i -= c)`.

"Those restrictions were added to make the for method design rule more readable.". Why did we add the restrictions that `begin` $<$ `end` and `begin` $>$ `end` to ascendant and descendant for loops, respectively? Because, internally the framework can assume, based on those restrictions and the ones formalized by OpenMP, that every time a loop has `begin` $<$ `end` is a loop semantically equivalent to:

`for(int i = begin; i < end; i += step)`

and that when `begin` $>$ `end` is a loop semantically equivalent to:

`for(int i = begin; i > end; i -= step)`

Otherwise, if the framework only relied on the following two OpenMP formalized restrictions:
If $\beta \in \{\jmath < \lambda, \lambda > \jmath\}$ then $\gamma \in \{\jmath$ `+= c`, $\jmath$ `= `$\jmath$ `+ c`, $\jmath$ `= c + `$\jmath\}$;
If $\beta \in \{\jmath > \lambda, \lambda < \jmath\}$ then $\gamma \in \{\jmath$ `-= c`, $\jmath$ `= `$\jmath$ `- c`$\}$.

the framework would have to use either the conditional operator ($<$ or $>$) or the loop step to figure out if the loop is ascendant or descendant. Using the conditional operator would be difficult since the framework is intercepting methods and reading its arguments; how would the programmer express the conditional operator through method arguments ?! Using the loop step would mean that the programmer would have to pass its sign in the method call. This means that, for example, with a loop of the form `for(int i = 11;i > 0;i -= 1)`, applying the *for* method design rule would result in the code presented in Figure 5.33. We felt that, although reading line 2 of Figure 5.33 looks relatively natural, we could not say the same about line 6. Having `i += step` when in reality the loop is decreasing in each iteration seems strange.

```
1 void code(){
2   loop_method(11, 0, -1);
3 }
4
5 void loop_method(int begin, int end, int step){
6   for(int i = begin; i > end; i += step){
7     // some code
8   }
9 }
```

Figure 5.33: *For* design rule : Extracting the loop body using explicitly the step sign.

Finally, the necessity of transforming $\leq$ and $\geq$ relates to the fact that the conditional operator is not passed as a parameter and, unlikely OpenMP the framework does not rely on static analyses to read the loop body. The framework needs to know the conditional operator to determine the number of iterations of the loop to be parallelized. Therefore, we decided to remove ambiguousness and stick to $<$ and $>$ conditional operators.

The second loop form supported by the framework can only be applied over collections that implement the *java.util.List* interface because it provides methods such as *get(int index)*, *set(int index, E element)* and *add(E e)* that are useful for the framework internal mechanisms. Furthermore, the collection that the loop is iterating over has to remain immutable during the loop execution otherwise it might lead to race conditions, among others.

5.4.1.2 Data-related design rules

To understand the design rules for data-related transformations (DRT) presented in Section 5.3.3, first, it is necessary to understand what each DRT intend to do. As with the computational transformations, the DRT of our SM and DM libraries are inspired in the OpenMP and MPI, respectively. Without entering into too much detail, the DRT of SM are related to making shared data private, whereas the DM is related to *sharing* data through processes communication.

The DRT from OpenMP 3.1 are mostly centralized around primitive local variables and are not as useful when it comes to objects and their fields. Oppositely, MPI centralizes mainly in objects. Our framework faces the opposite *problem* of OpenMP. AspectJ can deal with objects and their fields but not with local variables. This problem is especially problematic for primitive local variables since that in addition to AspectJ not being able to intercept them there is no explicit memory reference to these variables.

Concerning local objects, AspectJ can intercept their creation, but can neither differentiate between them nor intercept direct accesses to them[17]. That happens because AspectJ is not able to reference them by name, unlike a field. Therefore, given this situation, a possible design rule could have been to force the programmer to make all the data to be transformed into an object field. That would mean that users would have to turn both primitive and object local variables into fields. However, in our approach, DRT are tightly-coupled to a particular computational code transformation represented by a method call (labeled by us as *DRT method*). Hence, we use the argument of the DRT method to pass the data to be transformed. Unfortunately, with Java, it is not possible to pass a reference to a primitive local variable. Therefore, programmers need to either turn these variables into objects or fields. The former although not providing AspectJ with a directly interceptable variable, it provides a memory reference to that variable. Combining the memory reference with the possibility of passing it as a parameter of a method provides the framework with more than enough context to correctly perform the desired code transformations. Passing the argument as a parameter combined with specifying its position with the pointcut *args* allows to identify and distinguish between local objects. The memory references give the framework the freedom to update these objects content. Consequently, no additional design rules are required to apply DRT to local objects.

[17] We are referring to the object itself as an entity, not its fields. AspectJ is still able to detect reads and writes of the local object fields (if any).

We could have generalized the design rule for all DRT and stated that local primitive variables should be turned into either fields or local objects. Nevertheless, this raises the question: *"Do all DRT need this design rule?!"*. To answer that question we present a case-by-case analysis. For simplicity let us consider only the most used SM DRT, that can be applied to local variables, namely: *private*, *firstprivate*, *last private* and *reduction*. We can further divide these DRT into two groups, based upon the relevance, after the DRT method, of the value of the variables used in the DRT. In the first group (*private* and *firstprivate*) that value is irrelevant, whereas in the second group (*lastprivate* and *reduction*) that value is relevant. To better visualize the upcoming scenarios let us use the code of Figure 5.34 and assume that the user wants to apply DRT to variables that will be used inside the *loop_method*.

```
1  void method(..) {
2  // Block A
3  loop_method (0 , 10 , 1, ...);
4  // Block C
5  }
6
```

Figure 5.34: Example of data-related code transformation.

If the user wants to apply DRT into the local primitive variables in Block A, these variables had to be passed as arguments of a *DRT method* (line 3). Therefore, these variables will always be private to threads, because either they are created inside a parallel region or are passed as the argument of one. Hence, in our approach, there is no point in applying the *private* or *firstprivate* into local primitive variables, because their value will be passed as a parameter of the *DRT method*, and that value is irrelevant outside the *DRT method* call.

In constructors such as *lastprivate* and *reduction* it is implicit that the values of the variables, on which these constructors are applied, are relevant outside the *DRT method*. Thus, these primitive variables have data dependencies, before, during and after the *DRT method* call. That implies that during the method refactoring for the computational transformation these dependencies were already solved (shown in Figure 5.31), either by creating local objects or object fields. In both cases, the framework can deal with them without further design rules.

From the SM case-by-case analyze it is possible to conclude that, in practice, to apply the AOmpLib DRT the user does not need to care about turning primitive local variables into objects or fields. From the DM side, it is necessary to turn primitive local variables that will be used in the AOdmLib constructors into objects, or fields.

5.4.2 Shared Memory Library

5.4.2.1 Execution model and computational transformations

In programs parallelized with AOmpLib, the execution starts with a single thread, designated master, that intercepts the *main* method of a given project (specified by the user). Before executing the main, the master creates all the necessary AOmpLib data structures (*e.g.*, thread pools), reads the configurable options of the library and generates unique IDs for the threads in the pools. The master has the ID zero, and the other threads have an ID ranging from one to the maximum of threads defined minus one. These IDs are particularly useful in the *for* constructors, since the loop distributions are calculated based on them. Moreover, these IDs can be explicitly accessed by the users as well, which can be helpful to fine-tuned parallelizations (*e.g.*, tuning loop distributions).

Internally, the AOmpLib uses *hashmaps* to map the workers to their respective data structures (*e.g.*, team ID and private data). However, if nested parallelism is enabled a more complex structure is used since a worker can belong to multiple teams and a team can have several workers. Thus, additional logic is required to keep track of each worker and its respective teams, which results inevitably in higher overhead compared with a non-nested parallel region environment [Par, DHP08]. For these reasons, and since OpenMP does the same, nested parallel regions in AOmpLib are disabled by default.

When the thread master encounters a parallel region, it will request a team of workers from a *regular* thread pool. If the master encounters a *task* constructor, it will request workers from a *ForkJoinPool* thread pool instead, since this pool is more suitable for that kind of work (*e.g.*, divided-and-conquer algorithms) [Oak14]. Unlike the pool used for the parallel regions, *ForkJoinPool* allows workers to suspend tasks in favor of the execution of other tasks [Oak14]. Furthermore, as with the TBB sophisticated scheduler, workers of that thread pool have their (own) task queues and are allowed to steal work from the other queues [Oak14]. This work-stealing approach makes this pool suitable for applications with load balancing problems. However, it also comes with an overhead higher than *regular* thread pools. Furthermore, such a pool is not the most suitable to deal with static and dynamic loop distributions. Static scheduling predetermines the amount of work that workers will perform without having to resort to sophisticated locking. For the dynamic scheduling, it is possible to use an approach with lower synchronization overhead than a work-stealing mechanism with multiple queues.

In AOmpLib, a parallel region is the context of a method call, which will be intercepted by the parallel region annotation/pointcut (line 1 of Figure 5.35). After the thread master requested a team of workers, each one of them will also call the intercepted method (line 8) and implicitly synchronize at the end of its execution (line 10) unless the *no_wait* option was enabled. The number of workers in a team is calculated (line 5) in a similar way to OpenMP own heuristic[18]. Figure 5.35 shows a simple code snippet of the parallel region implementation. Naturally, this implementation needs to take into account if the nested parallel regions are enabled, the creation of the necessary data structures and so on. For instance, it is required to create a barrier *per* team to make sure that workers will synchronize before resuming the parallel region. If nested parallelism is disabled, for performance reasons, that barrier is created once at the beginning of the application with a default number of workers *per* team in mind. That barrier is created again only if the number of workers changed between consecutive parallel regions.

```
1  public abstract pointcut parallel();
2
3  void around() : parallel() ... {
4    ...
5    final int totalOfWorkers = requestNumThreads();
6    assignTasksToTeam(totalOfWorkers, () -> {
7      /** Task for the workers **/
8      proceed();
9      if(!no_wait()) {
10       /** call team barrier **/
11     }
12   });
13   /** Master task **/
14   proceed();
15   if(!no_wait()) {
16     /** call team barrier **/
17   }
18 }
```

Figure 5.35: AOmpLib - Code snippet of the parallel region implementation.

Figure 5.36 shows part of the code that performs the dynamic loop distribution. To keep track of the task to be assigned this constructor uses a lock-free strategy that relies on Java *AtomicInteger* class. Unfortunately, it was not possible to make the entire code of the dynamic loop constructor lock-free. To avoid race conditions, at the beginning of that constructor we used barriers around the thread master initialization of the *AtomicInteger* object (initialized with the first iteration of the loop). This mutual exclusion region was necessary because the *for_dynamic* annotation/pointcut can intercept the same method multiple times during the same application execution. Thus, it is vital to reset the atomic iteration counter in-between those method calls. This constructor is a good candidate to be improved in the future (*i.e.*, reducing its overhead).

[18]The complete algorithm can be found on pages 36 and 37 of [apiva]

```
1  ...
2  void around(int begin, int end, int step) : for_dynamic()
3                                                && args(begin, end, step, ..){
4    ...
5    final int chunk = chunk();
6    /** for(i = begin; i < end; i += step) **/
7    if(begin < end)
8    {
9      final int chunkSize = chunk * step;
10     int currentIteration = atomicTask.getAndAdd(chunkSize);
11     while(currentIteration + chunkSize < end)
12     {
13       proceed(iteration, iteration + chunkSize, step);
14       currentIteration = atomicTask.getAndAdd(chunkSize);
15     }
16     /** Dealing with the remaining iterations **/
17     ....
18   }
19   else if(begin > end)  /** for(i = begin; i > end; i -= step) **/
20   {
21     ...
22   }
23   ...
24 }
```

Figure 5.36: AOmpLib - Code snippet of the dynamic parallel *for* implementation.

After having deduced if the loop to be dynamically parallelized is ascending (line 7 of Figure 5.36) or descending (line 19), each worker can request chunks of loop iterations to compute. Each request is performed in an atomic manner using the *getAndAdd* method (lines 10 and 14) from the *AtomicInteger* class. Since the remainder of the division of the total of iterations of the original loop by the chunk size may not be zero, we divided the loop iterations into two groups. The first group is executed in parallel, and contains all the iterations starting from the beginning of the loop up to the maximum number of iterations that divided by the chunk size has a remainder of zero (lines 10 to 15). The second group contains the remaining iterations and is executed sequentially (line 16). This heuristic could have been simplified by calculating everything inside the first loop (lines 10 to 15) and ensuring that it did not surpass the original *end* variable. However, that would lead to extra conditional logic added to every chunk of iterations assigned. Hence, for performance reasons, we discarded that approach. Especially, because not only is the code from the first loop executed in parallel (lines 10 to 15), but also the iterations are assigned in a non-deterministic matter – making it very unlikely that the compiler could mitigate the extra conditionals.

5.4.2.2 Data model and data-related transformations

For the most part, AOmpLib computational-related constructors strictly follow OpenMP 3.1 standard. However, regarding the DRT there are some notable differences. These differences come from the Java model, AspectJ limitations regarding local variables and the difference between the strategies used by AOmpLib and OpenMP to perform the code transformations.

The OpenMP 3.1 DRT are mainly intended to be used on primitive local variables. The programmer is not allowed to use OpenMP *shared* or *private* data clauses in variables that are of a reference type. For class instances, it is necessary to provide the class copy constructor [apiva]. Regarding reduction operations, the OpenMP 3.1 standard [apiva] states that: *"Aggregate types (including arrays), pointer types and reference types may not appear in a reduction clause"*[19]. AOmpLib DRT work almost contrarily to OpenMP, once it allows the use of fields of classes as well as reference types, but restricts the use of local variables of primitive data types. Unlike AOmpLib, in OpenMP 3.1, it is not possible to apply DRT (with exception of *threadprivate*) in class fields (or structure). This inability to deal with fields can lead to situations that require a significant amount of code refactoring. Fortunately, to apply DRT in fields, AOmpLib only needs the name and scope of the fields to be transformed.

We explain next in more detail the type of DRT constructors offered by the framework, namely *method pointcuts*, *set/get pointcuts* and the *private object pointcut with annotations*.

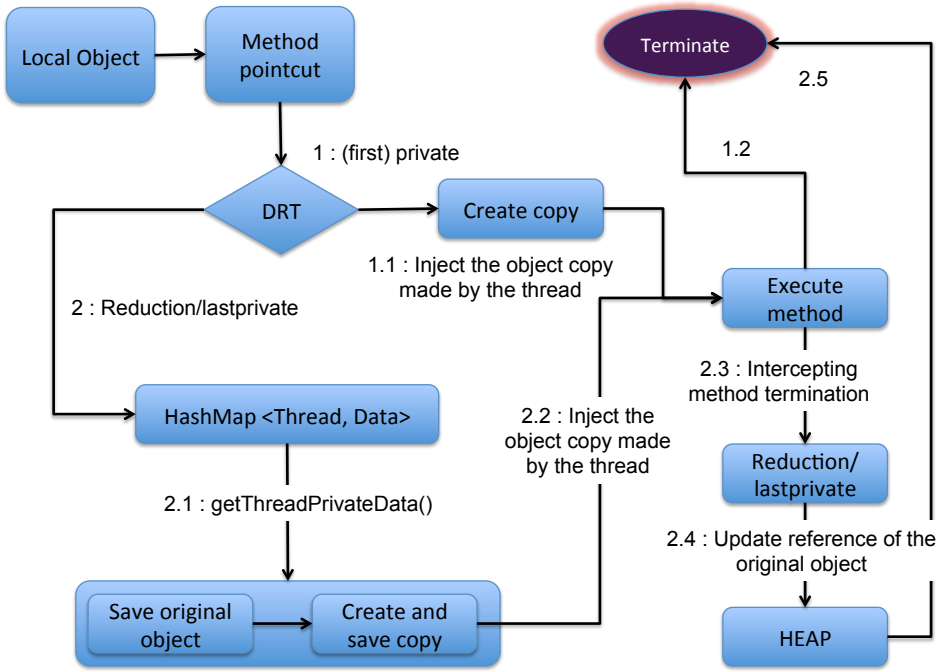


Figure 5.37: AOmpLib - Overview of a DRT using the *method pointcuts* approach.

[19]Currently, with OpenMP 4.5, it is possible to perform, for example, reductions of arrays using additional annotation clauses parameters.

Figure 5.37 presents a high-level diagram of how the *method pointcuts* DRT works. For example, when applying a reduction into a local object passed as an argument of the *DRT method*, each thread intercepts that argument, saves the reference to the local object and replaces the *DRT method* argument with a newly created object copy. At the end of the *DRT method*, all the object copies made by the threads are reduced into a single object. Finally, the thread master is responsible for updating the original object content.

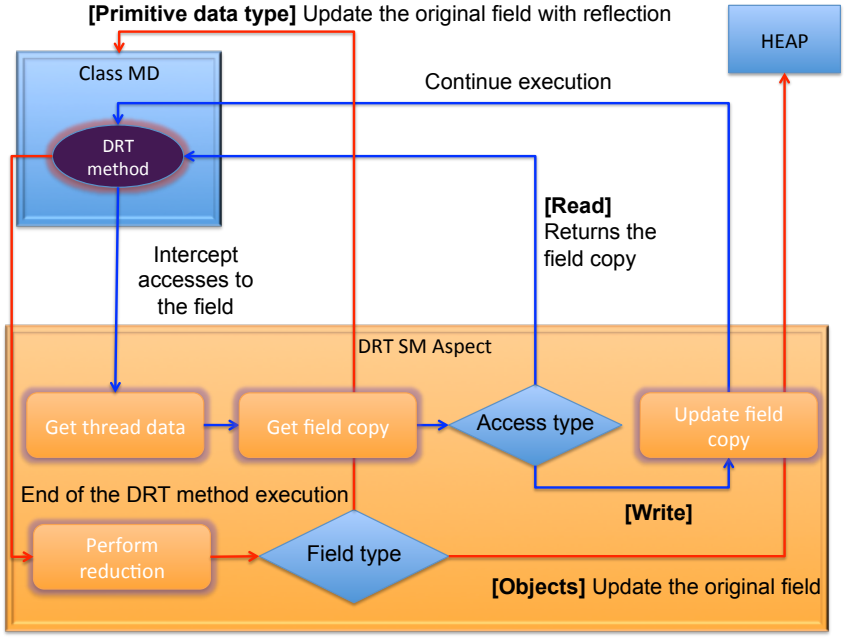


Figure 5.38: AOmpLib - Overview of a DRT using the *set/get pointcuts* approach.

Figure 5.38 shows an overview behind a DRT (in this case a sum reduction) applied to the fields of a shared object, using the *set/get pointcuts*. With AOmpLib it is possible to intercept the arguments or the target object of a method and replace them by thread private copies. However, it is impossible to do the same with fields of a shared object. Because the object is shared, its fields are also shared among threads. Nonetheless, the AOmpLib can, with the help of *set/get* pointcuts, intercept every access to those fields.

Using Figure 5.38, let us imagine that the user wants to reduce the field *epot* of the class *MD*. Internally each thread will create a copy of that field and initialize it with the appropriate value (*i.e.*, 0 in case of a sum reduction). During the scope defined by the user, the AOmpLib will intercept all accesses to *epot* and, for each one of them the threads will request their *epot* copy from their private data. If the access type is a read, the value of the thread *epot* copy is returned to the base code, otherwise (*i.e.*, a write), threads will update their *epot* copy with its new value and resume execution. At the end of the *DRT method* execution threads will reduce all their *epot* copies into a single one. If the variable *epot* is of primitive data type, then the thread master will directly update, using reflection, the original *MD epot* field with the result

of the reduction. If the variable *epot* is an object instead, then the thread master will directly update its value on the heap. The reason why the same cannot be done to the primitive data types is that it is impossible to access them through an explicit memory reference. Nevertheless, because internally AOmpLib has the memory reference to the *MD* object and the name of the fields to be changed, with the help of reflection it is possible to update its value. The *set/get* pointcuts do not help in this case because there is no reasonable and efficient way of knowing when it will be the last access to the variable *epot* to inject, at that time, the reduction result.

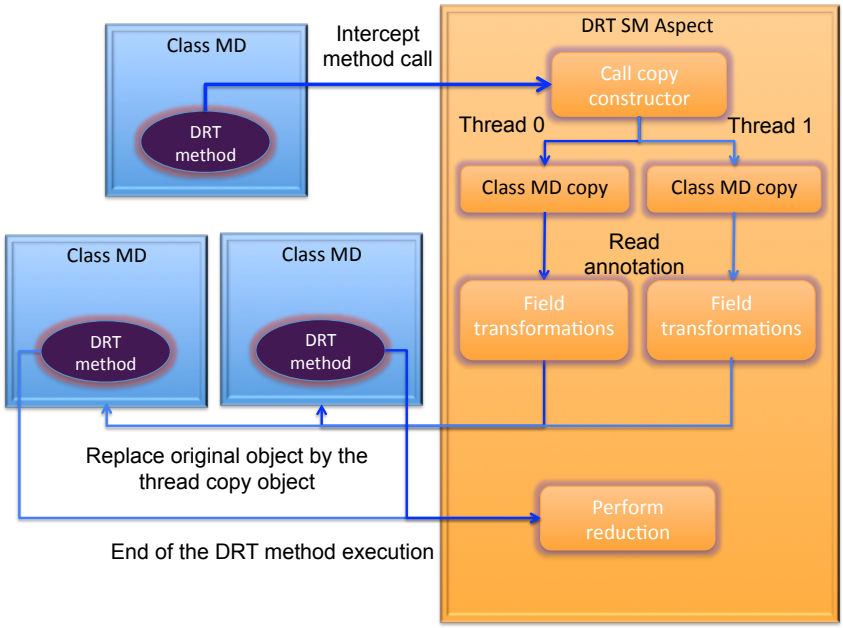


Figure 5.39: AOmpLib - Example of a DRT using the *private object pointcut and annotations* approach.

Taking the example of Figure 5.38 but instead of using the *set/get* pointcuts in the shared object, the *private object pointcut and annotations* approach was used. The workflow for that example would be something similar to the one shown in Figure 5.39. In the *private object pointcut and annotations* approach a copy of the *MD* object for each thread is created (two threads in the illustration of Figure 5.39). Afterward, each thread reads the annotations from its private copy and applies the necessary transformations to its fields. Then right before the execution of the *DRT method*, the original *MD* object is swapped by the threads private copy. Because each thread has a copy of the *MD* object, it is not necessary to intercept all the calls to the field *epot*. Instead, threads directly access their private field *epot*. Because during the *DRT method* execution no field interception is needed, annotations can be used to define the DRT to be used. This approach provides potentially better performance than using *set/get* pointcuts. When the AOmpLib reads those annotations, logic will be automatically inserted, before and after the *DRT method*, to perform the DRT correctly.

5.4.3 Distributed Memory Library

In Java, it is not possible to handle memory pointers explicitly. This particularity makes building the messages to be exchanged among processes more cumbersome and complex in MPI for Java than for C. For instance, in Java to reduce a matrix the user has to call the MPI reduce routine for every line of the matrix, whereas in C only one call to the MPI reduce routine is needed – as long the matrix was allocated contiguously in memory. All of these technicalities are hidden behind AOdmLib high-level constructors that exploit Java method overload capabilities to extend the MPI functionality while keeping the constructors with the same name.

AOdmLib has static, user-defined, and dynamic loop distributions. The heuristics of the first two are the same as the ones used in AOmpLib but replacing thread ID by process ID. The dynamic distribution, however, works differently from the one used in AOmpLib. The dynamic distribution of AOdmLib uses an approach similar to the task-farming model, where the master process assigns and keeps track of the loop iterations. The master will send chunks of iterations to the remaining processes on-demand. When there are no more iterations to be consumed, the master will send to every process a special message notifying them about the current status, so that all processes can normally resume their execution. It is expectable that the dynamic *for* of the AOdmLib has a higher overhead compared with the one from AOmpLib.

```
1  static aspect splitRowsMatrixA extends DM_Share <MM> {
2    ...
3    @Override
4    public void data(MM baseClass) {
5      baseClass.A = create_view(baseClass.A);
6    }
7  }
```

Figure 5.40: AOdmLib - The method *data* from the data partitioning module.

AOdmLib needs the method *data* along with the code from line 5 of Figure 5.40 to switch between local and global views at any time. To change the current view, AOdmLib must change the memory reference of the partitioned data from the base code (*e.g.*, matrix A). Therefore, whenever the AOdmLib needs to change between views, the method *data* is called internally, and the new view is returned to the base code using the returning value of the method *create_view*.

Aside from a few exceptions (*e.g.*, barrier), the majority of OpenMP constructors are applied around a block of code (*e.g.*, a parallel region). The scope of the MPI constructors is slightly different from those of OpenMP, partially because of the differences between SM and DM models. Compared to OpenMP, MPI relies more on single point constructors (*e.g.*, communication routines). Since AOdmLib is based on MPI, we decided to offer an alternative approach to the one of AOmpLib, less depending on pointcuts to express the code transformations. This approach relies more on method calls than pointcuts, which enables users to express the communication routines more easily. Furthermore, this approach is closer to the way MPI is usually used.

We will use the example in Figure 5.41 to showcase the advantages of the method call approach compared with the pointcut approach, in the DM environment. Let us assume that we had adopted the AOmpLib approach (*i.e.*, using pointcuts) to the DM environment as well. For instance, to reduce variables, we would have to specify with pointcuts the variables to be reduced, and when. Moreover, we would have to use method overriding to define some of the reduction options (*e.g.*, operation and root process). Let us assume that the user wants to reduce the variable x of Figure 5.41 right after the call of the *method1* (line 12). Doing such action with pointcuts can be tricky. Firstly, *set/get* pointcuts "have to" be used to intercept that field variable[20]. Secondly, it is necessary to find the first join point in the base code after the call of *method1*, where the variable x is used (line 9). Additionally, the user would have to ensure that this join point is only executed once so that multiple interceptions are not triggered by AOdmLib. This join point is needed to specify a pointcut that triggers the reduction of x among processes and returns the result to the base code. Ideally, the user would rather perform the reduction at the end of line 12, however, because at that time there is no access to the variable 'x' the user has to postpone this interception.

```
1   private int x;
2   Random random = new Random();
3
4   void method1(){
5     for(int i = 0; i < N;  i++)
6       x += random.nextInt(N);
7   }
8   ....
9   void method2(){x = ...;}
10  ...
11  void method3(){
12    method1();
13    method2();
14  }
```

Figure 5.41: AOdmLib - Example to showcase the advantages of method call *vs.* pointcuts.

An alternative approach would be to create an aspect with two pointcuts for every communication that the user intends to perform, one that only intercepts the data (*i.e.*, field), and another to define the point where the communication should be performed (*e.g.*, after *method1* call). The first pointcut would provide the library with the name of the variable and possibly its reference (if it is an object). The library could then save this information to later used it to reduce the variable after executing the second pointcut. Unfortunately, this approach has some drawbacks. First, it is necessary to provide the library with an access to the field. For performance reasons, the join point provided by the user should be only intercepted once and before the call of *method1*. For example, if the user would define the following pointcut *getVar()*

[20] A combination of design rules with the *call* pointcut can be used instead of *set/get* pointcuts.

: set(... x) && withincode(... method1()), according to the code of Figure 5.41, the AOdmLib would intercept `N` times the field *x*, even though one would be enough. The second issue is that the user may have to define every parameter of the communication using method overriding. The problem of finding the correct join point is not so severe in the AOmpLib because the SM code transformations and DRT are tightly-coupled around a given method call. Hence, the scope of these code transformations is well-established. Moreover, in the SM DRT, the use of *set/get* pointcuts made more sense since AOmpLib wants to intercept and replace all accesses to a given field within a certain method execution, not just to inject, in a single moment, the result of an operation (*e.g.*, reduction). Because of these issues, we provide more convenient approaches for the use cases of communication routines and splitting/gathering of data.

```
1  static aspect Communication extends Dm_Comm <MM>{
2    pointcut comm_after() : call(... matrixMultiplication(...));
3
4      @Override
5      public void data (MM mm){
6        allReduce(mm.B , Reduction_DM_OP.SUM));
7      }
8  }
```

Figure 5.42: AOdmLib - Example of an *all reduce* using method call approach.

Figure 5.42 presents a code snippet that will be used to explain in more detail the method approach. With this approach, the user creates an aspect with one pointcut. That pointcut declares the point in the code where a set of communications will occur. In the example of Figure 5.42, the communication will happen after the *method1* call (line 1). If the user wants the communication to take place before the method call the pointcut *comm_before* can be used instead. The *data* method receives as a parameter the object that contains the code of the method defined in the pointcut of line 1. This argument is helpful since it provides easy access to the object context. Worth noting that the *data* method is an abstract one, thus when using the communication aspect the user will be automatically reminded to implement that method. In line 6, the user expresses clearly the communication to be used, its data and options. Without the method approach, this single line of code would have resulted in several pointcuts and method overrides. Especially, if the user wanted to perform several other communications calls at the same point, instead of a single *Allreduce*. In practice, with the method approach, AOdmLib turns the code of Figure 5.41 into the one shown in Figure 5.43. From the latter code, it is possible to see that legacy code of that form can be seamlessly parallelized by AOdmLib. The user merely has to extract the MPI related concerns from the base code of Figure 5.43, use the AOdmLib features presented in Figure 5.42 and will end up with the code of Figure 5.41.

```
1  void method3(){
2    method1();
3    data(this);
4    ...
5    method2();
6  }
7  public void data(T baseCodeObject){
8    allReduce(baseCodeObject.x, Reduction_DM_OP.SUM);
9  }
```

Figure 5.43: AOdmLib - The end result of apply the method call approach.

Sometimes the data to be exchanged among processes is not a field but rather a local object. Clearly, in that case, the method approach is not suitable. Nevertheless, the user can either turn the local object into a field or use the AOdmLib appropriate pointcuts. The local object is also handleable since it is passed as an argument of a method. The user can specify the data using the *args* pointcut, as with the AOmpLib.

Chapter 6

Validation and Results

This section evaluates the performance and programmability of our framework in comparison to alternative approaches. To evaluate and validate the framework we used a set of nine case studies, namely a molecular dynamics simulation (MD), a Java-tuned matrix multiplication (MM), and a group of seven case studies of the *Java Grande benchmark suite* (JGF) [SBO01].

We can divide the case studies used into two groups. The first group includes the case studies on which a significant amount of time was spent improving their sequential code and tuning their parallelization, namely the MD (based on the JGF implementation) and the MM. In this group of case studies, we describe in greater detail the design rules, pointcuts, and parallelizations developed. The second group has seven algorithms from the JGF (including their sequential, SM and DM versions). This group was used to study how the framework deals with a broader range of case studies and with legacy code. The MD of the first group along with the algorithms of the second group corresponds to all parallel benchmarks from the section II and III of the JGF. To evaluate the performance of our framework we measured the overhead of:

1. Design rules;
2. SM layers built with AOmpLib;
3. DM layers built with AOdmLib;
4. Hybrids built by composing the SM and DM layers.

In the first point, we compare the execution of the base (sequential) code with and without design rules, to understand the overhead that these rules may introduce. In the base code with design rules, we included all the rules needed to enable the SM and DM parallelizations. In the second point, we compare our SM library with intrusive SM Java[1] and with JOMP implementations. In the third point, we compare our DM library with Java MPI implementations[2]. Lastly,

[1]Both the SM library and the intrusive implementations use the Java concurrency mechanisms.
[2]Both the DM library and the intrusive implementations use the same MPI version.

for the case studies with good scalability (*i.e.*, those with speedups over 15.0× in their SM and DM intrusive implementations), we also benchmarked their hybrid parallelizations. These hybrids combine the best SM and DM versions of a case study, with as few as possible changes to both versions. These hybrids were built, mainly, to verify how easy it is to develop them with our framework. Ideally, it should be only required to compose the SM and DM layers. Finally, we compare the performance of the hybrids with pure MPI versions.

Optimizing an application is a hard task; especially parallel ones since the programmer has to consider aspects such as load balancing, synchronization overhead, the trade-off between legibility and performance, among others. Performing such tasks intrusively makes matters worse. Typically, to test the different parallelism-related approaches, the programmer will either repetitively modify the same base code or maintain multiple copies of it, one for each approach.

Our framework enables to quickly test different parallel optimization concerns by (un)plugging the correspondent modules. Hence, whenever possible, we exploited the capabilities of our framework to improve the parallel versions of the case studies. After employing the appropriate design rules into the base code, we took advantage of the properties of our layers (*e.g.*, pluggable and modular) to quickly, and conveniently, test different strategies (*e.g.*, loop distribution and data partitioning) aiming for a more efficient parallelization than the original one. Applying this prototyping procedure during the development of parallelizations is faster and easier than performing it in an intrusive manner, especially in the case of hybrids. Additionally, because C is a popular low-level language among the HPC community, for the majority of the case studies, we provide and measure C implementations of them[3]. These implementations include the sequential, SM OpenMP, DM MPI and hybrid versions (when it is the case).

Figure 6.1 depicts the workflow used to develop the different versions of the case studies. Deviations made to this workflow, for any case study, are documented. Firstly, we started with the original sequential version of the case study and examined if it could be improved. When that was the case, we also adjusted the corresponding original intrusive SM and DM versions to reflect these improvements. Secondly, we applied all the necessary design rules over the sequential version (improved or not) and developed the SM layer. When an intrusive SM parallelization was available (*i.e.*, all JGF case studies) our first SM layer mimicked that parallelization, otherwise (*i.e.*, MD and MM) we started with one that seemed reasonably efficient. During this step, several strategies were easily tested with our AOmpLib to improve performance as much as possible (step labeled as "SM Layer Improving Cycle" in Figure 6.1). Testing different strategies at this stage is just a matter of swapping different SM layers and, occasionally, adding some design rules, which requires less effort than intrusively changing an SM version. If an intrusive SM version already existed and a better strategy was found, we developed a new version (labeled "improved") based on the original version modified to implement the new strategy. In the cases

[3]Those C implementations (created by us) replicate, as much as possible, the best Java implementations.

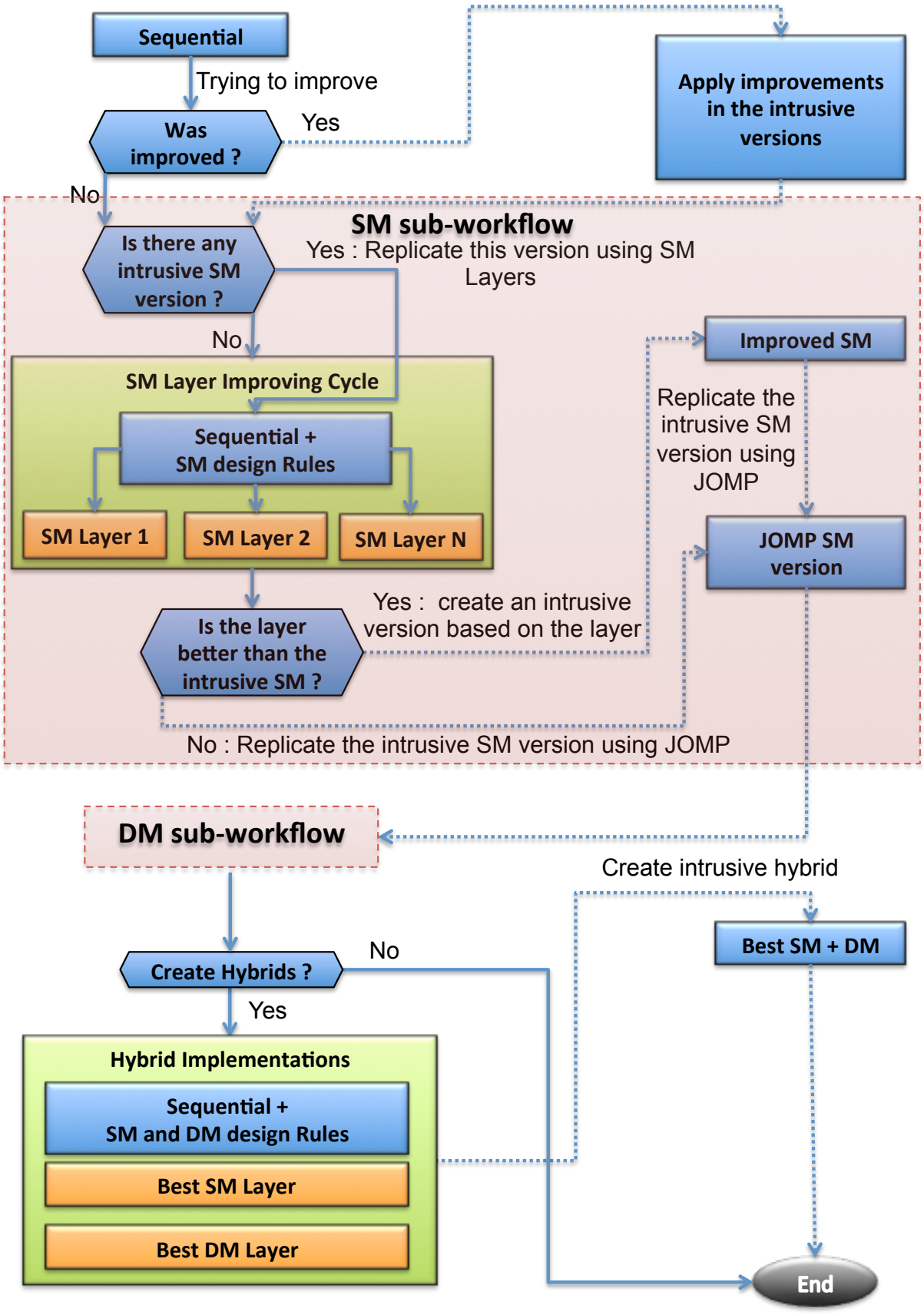


Figure 6.1: Workflow of implementing the different versions of the case studies.

that the intrusive SM version did not exist we created one based on the best version found with the layers. Afterward, we compared the execution times of the original, improved (if present) and AOmpLib SM versions. Additionally, a JOMP and a C implementation of the best SM layer were developed and compared. This process was then applied to the development of the DM versions (step labeled "DM sub-workflow"), however without a JOMP implementation. Whenever the best SM and DM Java intrusive implementations achieved speedups of 15.0× or more, we created hybrid versions out of them. Finally, after all the parallelization were developed, and tested, we examined the overhead of the design rules by executing the sequential version with all the design rules on it.

To measure the programmability, for each case study, we kept track of the parallel constructors used, the applied design rules, and the number of statements necessary with our approach *versus* the alternatives. Moreover, we also documented collateral side effects[4] of using the design rules, aspects, and pointcuts. Regarding the hybrid versions, we documented what were, if any, the difficulties of developing them with and without our framework.

For all the case studies, we decided to compose our SM/DM layers upon the same sequential base code. The goal is to understand how hard it is to use the same base code for all the different parallel versions. Finally, the AOmpLib and AOdmLib were used not only to try to find faster versions than the ones from the original case study but also to develop versions more readable.

6.1 Benchmark Environments and Methodology

We executed the tests in machines with two E5-2650 v2 processors[5] (NUMA), each processor with 16 physical cores and 32 logical with hyper-threading, 20 MB of shared cache L3 and a fixed frequency of 2.6 GHz. For the tests with multi-machines, we used a cluster with 8 of the previously mentioned machines corresponding to a total of 128 physical cores (256 logical). These machines communicate with each other using Myrinet up to a maximum of 8 processes *per* machine and Ethernet otherwise. All the used machines are from the *SeARCH* cluster[6] and run with CentOS 6.3, OpenJDK 1.8.0_20, OpenMPI 1.8.4 and GCC 4.9.3.

We performed 25 different measurements for the tests executed in a single machine and 10 for the tests executed in multi-machines. Afterward, we used the median of these measurements rounded off to the third decimal place. For all the parallel versions running in a single machine we tested them with 2, 3, 4, 6, 8, 12, 16, 24, and 32 threads or processes, whereas for those running in multi-machines we tested them with 8, 16, 24, 32, 48, 64, 96, 128, and 256 threads or processes[7]. In all the case studies we used a total of five different input sizes, from which 2

[4]This includes situations that we did not foresee when using our framework.

[5]`https://ark.intel.com/products/75269/Intel-Xeon-Processor-E5-2650-v2-20M-Cache-2_60-GHz`.

[6]More details about this cluster can be found in search.di.uminho.pt.

[7]For the (rare) case studies that, for some reason, did not follow this reasoning, we point out what was the number of threads or processes used.

or 3 came from the original JGF benchmark. Lastly, we aimed for inputs with sizes that fit in the cache levels as well as with sizes that only fit in RAM.

Since the versions implemented with our framework, JOMP, and intrusively are all Java-based, to avoid misinterpretations in the subsequent subsections of this section we use the terms "JOMP", "layer of aspects" (or simply "aspects"), and "Java" to reference implementations developed with our libraries of aspects, JOMP implementations, and (intrusive) implementations that only use Java, respectively.

The speedups of the JOMP, aspects, and Java implementations are calculated using the execution time of the same sequential version (*i.e.*., the fastest execution time of the sequential code with and without design rules). Regarding the speedups of the C implementations, we present two variants of them. In the first one, the speedups are calculated based on the C sequential execution time, whereas in the second variant (labeled "C adjusted") the speedups are calculated based on the fastest Java sequential execution time.

The gains of a given implementation (A) over another (B), regardless of the environment (sequential, SM, DM or hybrid), are calculated using the formula:

$$\textit{Gains } A \textit{ vs. } B = \frac{\texttt{execution time of B}}{\texttt{execution time of A}}$$

The calculation of the overall gains and speedups are the accumulation of the values from all inputs. For instance, the overall gain of Java *vs.* Aspects, in SM, would be the sum of the execution times of all inputs of the SM parallelization with aspects divided by the sum of the execution times of all inputs of the SM parallelization with Java.

Regarding the explanation of results we mainly, but not exclusively, focus on the overhead of the SM/DM layers of aspects and the design rules. Although we have results comparing C with Java, it is not the focus of this thesis arguing in favor of one language over the other. We believe that with enough time, in theory, one should be able to have a C implementation equal to, or faster than, a Java one. Notwithstanding, we wanted to understand what would be the performance difference between codes implemented as similar as possible in both languages, especially in the parallel versions. Before any performance analysis, we expected that C would be faster than Java, the intrusive Java implementations would be faster than the aspects and that our AOmpLib would be faster than JOMP. From the results that did not match these initial expectations, we only considered them worthy of an in-depth examination when their gain value is under 0.97. We use this approach because the tests were performed in an environment with several factors that may influence performance and are difficult to control (*e.g.*, the unpredictability of the Java garbage collector, the JIT compiler and the cluster itself).

6.2 Case Study : MOLDYN

Molecular dynamics (MD) is a powerful simulation method capable of simulating a variety of systems, both in and out of thermodynamic equilibrium [ALT08], and is mostly used to simulate the interaction among particles in a given environment. The MD used on this thesis – based on the MD from the JGF benchmark – performs a simulation for atoms of argon using the Lennard-Jones potential [Jon24]. Figure 6.2 presents a simplified execution call graph of that implementation.

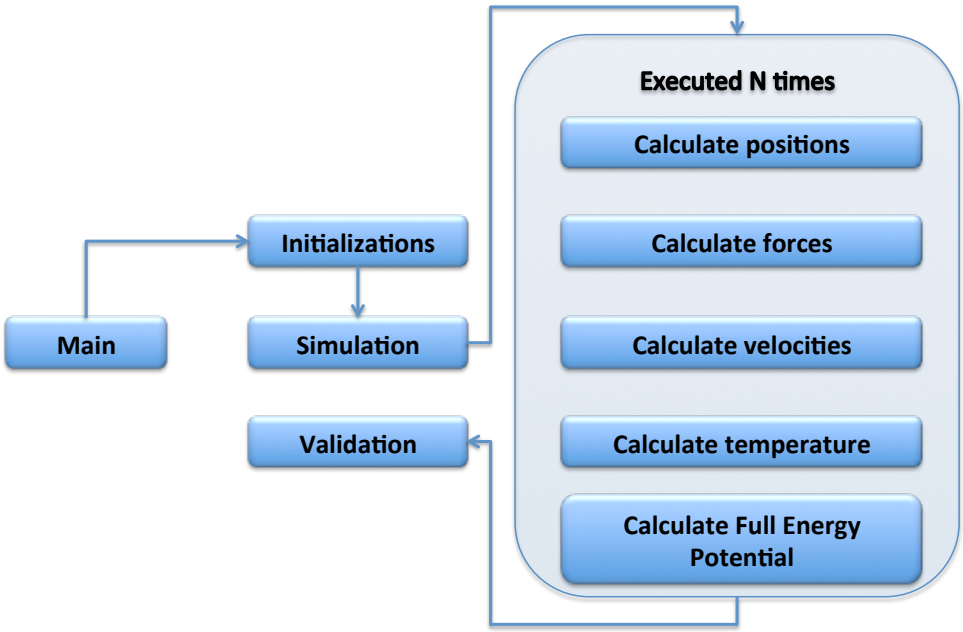


Figure 6.2: MD - The execution call graph (two levels deep).

In this implementation, as with most MD simulations, the majority of the execution time is spent in the step "Calculate forces" (detailed in Figure 6.3): for each particle, the total force exerted on it by the remaining particles as well as the distance between it and every other particle are calculated. Whenever the distance between two particles is less than a given value (labeled "condition" in Figure 6.3), their forces are updated along with some other variables related to the MD simulation. As shown in Figure 6.3 (second loop) the force calculation method takes advantage of the third Newton's law. This law states that the force that a body A exerts on a body B is equal, in magnitude, to the force that B exerts on A. Therefore, whenever two particles are within a given radius, the force exerted between them is calculated and both particles' total forces are updated. Consequently, reducing the number of iterations from $\mathtt{N}^2$ to $\frac{\mathtt{N}(\mathtt{N}-1)}{2}$. Although this optimization reduces the number of force calculations to half, and consequently the execution time, it introduces load balancing and data races challenges in the parallel versions, which increases their complexity.

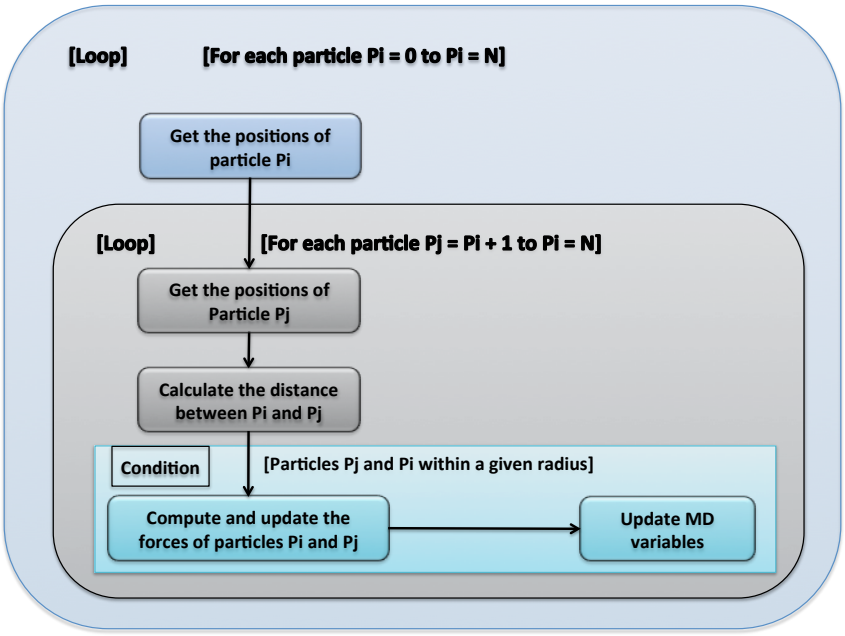


Figure 6.3: MD - The diagram of the sequential version of the force calculation.

The code from this case study was already presented in Figures 5.3 and 5.4 of Section 5.3.1. That base code included already some optimizations made by us. Before any parallelization, a careful analysis was performed, starting with the layout of the particles. The original JGF implementation uses an array of objects to represent the particles, where each object contains nine doubles to store the x, y and z components of the particles' force, velocity, and position. To improve the spatial locality, we changed the original particles layout to an object (*i.e.*, class *Particles*) containing nine arrays of doubles, one for each x, y and z components of the particles' position, velocity, and forces. Thus, the information of a given particle is accessible using the corresponding array index. This new layout was overall 1.13× faster than the original with the performance gain being most noticeable in the inputs that do not fit in the cache. The input sizes used in this case study are detailed in Table B.1 of Appendix B. The MD simulation executed only one iteration[8] for the two largest sizes (250k and 500k particles), whereas for the remaining sizes the original 50 iterations were performed.

A second improvement made was the decrease in the number of updates to the *MD* control variables (*i.e.*, *epot*, *vir*, and *interactions*). Instead of directly updating these fields inside the inner loop that calculates the forces between one particle and the remaining (line 18 of Figure 5.4), the updates are performed in local variables. Afterward, outside the inner loop (line 34), the *MD* fields are updated with the values of the corresponding local variables. This reduction in the number of accesses to the *MD* fields is particularly beneficial in the SM parallelization since the number of (potential) concurrent accesses of the threads to the shared data is reduced.

[8]The execution of a single MD simulation with 250k and 500k particles, using the original 50 iterations, would require more than 2 and 10 hours, respectively.

Identifying the parallel tasks and their granularity, and ensuring an equitable task distribution among threads/processes is paramount to achieve an efficient parallelization. Since the force calculation of all particles is the most computational demanding method[9], we analyzed this method extensively to find out the granularity of possible parallel tasks. This analysis showed that the smallest task granularity is the force calculation between a pair of particles. Inside that task there are two possible situations: 1) if the particles are within a given radius their force is computed and updated; 2) otherwise, there is no force computation and update. This granularity provides the highest number of tasks to be parallelized. However, this granularity is not coarse enough to overcome the parallelization overhead. At a higher granularity level, there is the force calculation of a particle with the remaining. This granularity level is the most suitable (in the architecture that we used) to be assigned to the threads/processes because it offers the best trade-off between the total number of parallel tasks and task granularity.

Since our MD uses the third Newton's law, we have to deal with data-dependency and load balancing problems. Regarding the data-dependencies, the arrays of forces (*i.e.*, *fx*, *fy*, and *fz*) of the object *Particles* and the fields *epot*, *vir*, and *interactions* of the object *MD* – fields updated inside the *updateControlVar* method – are the only shared variables rewritten inside the execution flow of the force calculation. Hence, these variables are susceptible to race conditions.

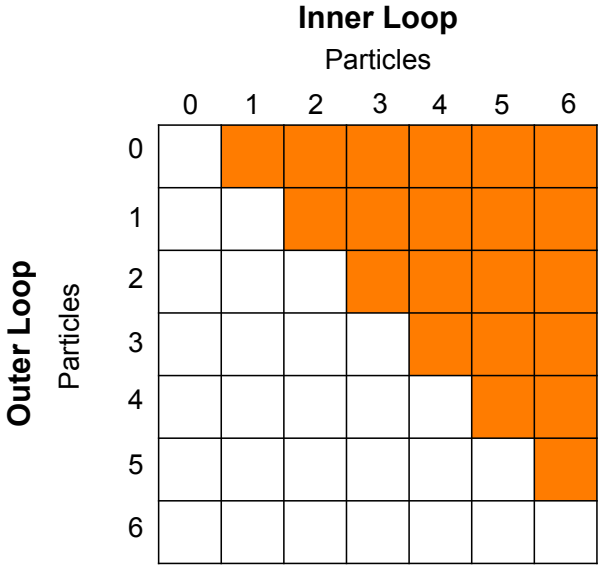


Figure 6.4: MD - Illustration of the force calculation between pairs of particles.

Concerning the load balancing issues, in Figure 6.3 we can see that the number of inner cycle iterations varies with the outer cycle iterations ($Pj = Pi + 1$ *to* $Pi = N$). Figure 6.4 illustrates this variation by presenting a matrix out of the force calculation iterations. The cells colored with orange represent the iterations that can be assigned to threads/processes, the

[9]For the different input sizes, the results have shown that the MD simulation spends 99% of its total time on the force calculation.

xx and yy axis represent the iterations from the inner and outer loops, respectively. Figure 6.4 demonstrates that evenly dividing the outer loop iterations among threads/processes would lead to serious work unbalance. For instance, if we split the outer loop iterations between 2 threads using the OpenMP default *for* distribution, the first thread would receive 15 pairs while the second thread only 6 pairs (*i.e.*, (4,3), (5,3), (6,3), (5,4), (6,4), and (6,5)). Hence, different static and dynamic distributions with different chunks sizes are worth being tested.

The next subsection details the workflow that leads to the final SM and DM layers of the MD case study. Moreover, we also compare the performance and programmability of each intermediate test layer to OpenMP/JOMP and intrusive implementations.

6.2.1 Shared Memory

Our starting point is the SM layer presented in Section 5.3.1 (Figures 5.7 and 5.9). That layer created a parallel region in the force calculation method and assigned to the threads the iterations of the force calculation between a particle and the renaming, in a round-robin fashion. Critical regions were used to deal with the data-dependencies. After ensuring the parallelization correctness of the SM layer, the next step was to improve its performance by first tackling the synchronization overhead and then the load balancing problems. In our proposed approach this corresponds to the development of a new SM Layer.

6.2.1.1 Dealing with synchronization overhead

The problem with the critical region is its high overhead, caused by frequently forcing threads to wait before executing that region, which increases with the number of threads and, consequently, limits scalability. With this approach, during the force update, threads wait outside the critical region even when they are not going to update the same particle. Using a lock *per* particle minimizes this problem by only forcing threads to wait for each other if they update the forces of the same particle. Another approach is to replace synchronization with data redundancy. In this approach, instead of updating the problematic fields directly in the *MD* and *Particles* objects, their values are accumulated in private copies assigned to each thread. At the end of the parallel region, the private values of all threads are reduced and afterward the master thread updates the original variables. The data redundancy strategy removes the synchronization among threads but requires the use of additional memory, which may impact performance. To decrease the synchronization overhead we tested the following approaches:

1. One lock *per* particle, and synchronizing the *updateControlVar* method with a critical region;

2. Data redundancy.

Locking approach

The first strategy required the creation of an array of locks, with a dedicated lock *per* particle. To create this array of locks and to (un)lock particles based on the index used to access the arrays of forces, AOmpLib needs to know both the size of the array to be created and, during the update of forces, the array position to be (un)locked. The size can be provided by applying the *object creation* design rule over one of the arrays used to save the particle's components (*e.g.*, fx[10]). The array position can be retrieved using the *method design rule* introduced during the creation of the critical regions (*i.e.*, *forceUpdate* method shown in Figure 5.8). Since the first argument of the *forceUpdate* method is the particle index, the array position to be (un)locked is already provided. Considering that no additional design rules were necessary, the next step was to define the pointcuts and connect them with the appropriate join points (Figure 6.5).

```
1 static aspect ParticlesLocking extends Sm_Locks
2 {
3   pointcut lock_array_creation()      : call(... double[] create_fx(int));
4
5   pointcut lock_array_access(int pos) : call(... void forceUpdate(int,...))
6                                         && args(pos, ...);
7 }
```

Figure 6.5: MD - Pointcuts to lock the accesses to the array indices.

With the pointcut declared in line 3 of Figure 6.5 the AOmpLib intercepts the creation of the *fx* array, extracts its size and creates an array of locks with the same size. The lines 5 and 6 indicate the method and its argument that contains the index of the array to be (un)locked, respectively. For every call of the *forceUpdate* method, the AOmpLib reads and uses its first argument (*i.e.*, the array index) to retrieve the correct lock from the internal array of locks.

OpenMP does not provide locks through annotations, but rather by explicit method calls to its API. Thus, the reasoning and total of statements needed to implement the locks with OpenMP or intrusively with Java are fundamentally the same. In a possible intrusive implementation, we could declare the array of locks as a field of *Particles* and initialize it in the constructor of that class (shown in Figure A.2). Afterward, the logic to acquire/release the lock associated with a particle would be placed around the code that updates the particles' forces (shown in Figure A.4). Our library provides the acquiring/releasing logic through the use of the first argument of the *forceUpdate* method and the *lock_array_access* pointcut. However, unlike the intrusive approach, our aspects neither introduce a new concern into the base code nor expose the locks to the outside. Our framework allows the user to seamlessly change the locking strategy by simply swapping between different layers while using the same base code. With an intrusive approach, to test different locking strategies (*e.g.*, locks and *critical*) the user has to either change the base code for each test or create different versions of the same base code.

[10]Figure A.1 from Appendix A shows the code of applying the design rule over the *fx* array.

Table 6.1: MD - The number of statements needed to implement the lock approach.

	AOmpLib	OpenMP C	Java Intrusive
Number of statements	6	11	14

Table 6.1 summarizes the number of statements needed for the locking approach (Figure A.3 presents a detailed explanation). For this layer, AOmpLib used, more or less, half of the number of statements used by the intrusive approaches (*i.e.*, OpenMP and Java). This difference comes mainly because most of the locking mechanism is handled internally by AOmpLib. Regarding complexity, in this case, our aspect layer is less complex than the intrusive approaches because the user only has to reason about how to pass the size to create the array of locks and the index to access these locks. The user does not have to deal with the low-level details of explicitly handling locks. Moreover, as with our aspects, in the intrusive implementations, the user also has to deal with the size of the array of locks and with the access to the appropriate lock from that array. Nonetheless, we acknowledge that directly accessing the array seems more natural than passing its index through a method parameter.

Data Redundancy approach

In the second strategy to deal with data races (*i.e.*, using data redundancy), we used constructors of the AOmpLib to provide private copies of fields of the *MD* (*i.e.*, *epot*, *vir*, and *interactions*) and *Particles* objects (*i.e.*, *fx*, *fy*, and *fz*). At the end of the parallel region, these private copies are reduced and the correspondent variables of the base code updated accordingly. For readability purposes, we named these fields as *problematic variables*. To create the private data, we tested two different AOmpLib approaches: 1) using *set/get* pointcuts combined with the *set/get performance* design rule (detailed in Section 5.3.3) to create private copies at the field level only; 2) creating private copies of the entire *MD* and *Particles* objects.

For the *set/get* constructors, we applied the *set/get performance* design rule only to the force fields (details in Figure A.5), since their accesses occur inside a double loop[11]. By applying this design rule, AOmpLib can inject the reference to the threads' private arrays into the local variables declared inside the force calculation method. However, due to this design rule, the forces fields were, inconveniently, passed as parameters of the method *forceUpdate*. This small inconvenient would not have occurred if we had started by testing the *set/get* pointcut before the *critical* constructor, because the *forceUpdate* would not have been created. Nevertheless, we could have simply inlined this method and tested the *set/get* pointcut effortlessly. However, in our opinion, the pros of maintaining the method outweigh the cons of removing it. Not only the method has made the code more readable, but it also has provided a useful join point to test different PRC strategies in the future. Notwithstanding, after finishing the layer testing

[11]The fields *epot*, *vir*, and *interactions* are accessed `N` times whereas the forces are $\frac{N(N-1)}{2}$, with `N` being the total number of particles.

workflow it is always possible to undo the unnecessary design rules. The aspects and pointcuts used to test the *set/get* approach are very similar to those previously presented in Figure 5.25.

```
1  aspect MD_SM extends SM_Layer
2  {
3    pointcut hotspot() : call(... void Particles.calculate_force(int,int,int, MD))
4
5    @SmData (List = {@OP(Type = Type.SUM, vars = {"fx","fy","fz"})})
6    static aspect Particles_Data extends Sm_PrivateObject <Particles>
7    {
8      pointcut private_object(PrivateObject particles) : hotspot() && target(particles);
9
10     public Particles Particles.copy() { ...}
11   }
12
13   @SmData (List = {@OP(Type = Type.SUM, vars = {"epot","vir","interactions"})})
14   static aspect MD_Data extends Sm_PrivateObject <MD>
15   {
16     pointcut private_object(PrivateObject md) : hotspot() && args(.., md);
17
18     public MD MD.copy() { ...}
19   }
20 }
```

Figure 6.6: MD - Pointcuts and aspects to create private objects.

Figure 6.6 presents the layer that creates, for each thread, private copies of the entire *Particles* (lines 5 to 11) and *MD* (lines 13 to 19) objects. The pointcuts in lines 8 and 16 replace, respectively, the target object and last argument of the *calculate_force* method by the appropriate thread private copies. The annotations in lines 5 and 13 indicate what fields should be reduced after the *calculate_force* method call.

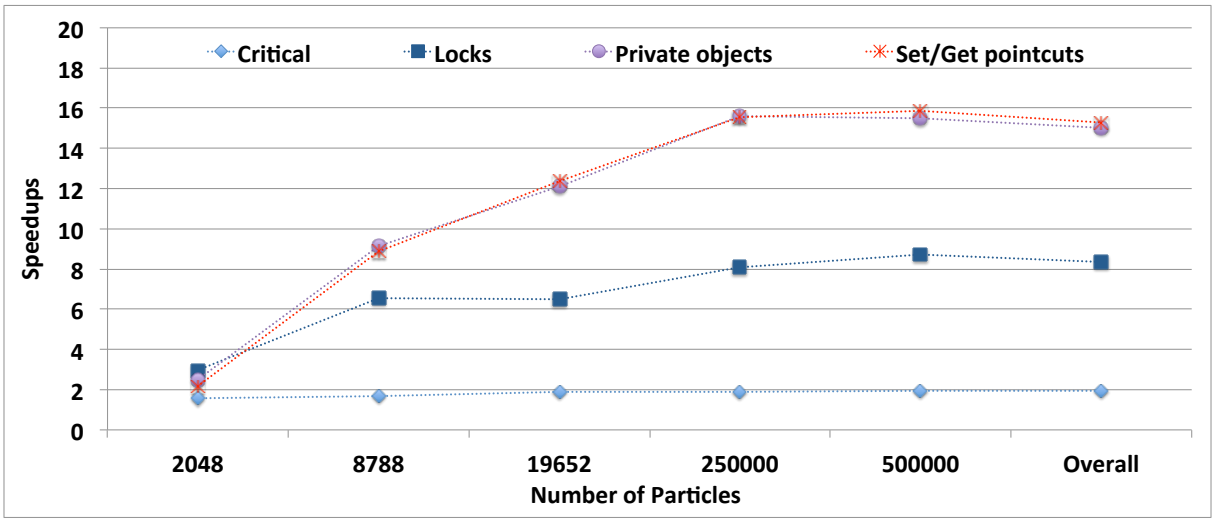


Figure 6.7: MD - The speedups of the strategies to deal with data dependency in SM.

The chart of Figure 6.7 and Tables B.2 (execution time), B.3 (scalability) and B.4 (max speedups) of Appendix B show the results of testing the different strategies to deal with data races. The data redundancy approaches were always the fastest, except for the smallest input size. For that size, having a lock *per* particle and synchronizing the *updateControlVar* method was the best approach. For the majority of the inputs, the *critical* approach stops scaling after 4-6 threads, the *locks* after 12-16 threads, and the data redundancy after 24-32 threads. In total, for all the inputs, the *set/get* approach achieved an overall speedup of 15.27× closely followed by the private objects approach with 15.03×. Since both approaches achieved almost the same overall speedup, we have chosen the more readable of the two, in this case, the private objects approach. Nevertheless, due to the flexibility of our layers, we could have created a SM parallelization that would use for each input their fastest layer. Hence, for the smallest input, we would use locks *per* particle and *updateControlVar* method synchronization, for the second and fourth inputs *private objects* and for the remaining inputs the *set/get* approach.

```
1 public class MD {
2   private ThreadLocal <ThreadPrivateData> mdThreadPrivate = ...;
3   ...
4   protected updateControlVar(double e, double v, int i){
5     mdThreadPrivate.get().updateControlVar(e, v, i);
6   }
7   ...
8 }
```

Figure 6.8: MD - Code snippet of the approach with *ThreadLocal* variables.

To intrusively replicate the *set/get* approach, the accesses to the *problematic variables* would have to be replaced by accesses to the corresponding thread private variables. Using *ThreadLocal* objects (line 2 of Figure 6.8) or creating arrays with the variables to be replicated among threads (line 3 of Figure 6.9) are two possible strategies to mimic the *set/get* approach.

```
1 public class Particles {
2   private final double fx[],fy[],fz[]; // forces
3   private final double fxTD[][],fyTD[][],fzTD[][]; // thread forces
4   ...
5   private void forceNewtonsLaw(final MD md, int pA, int threadID){
6     ...
7     // Inject private thread references
8     final double fx[] = this.fxTD[threadID];
9     ...
10    for (int pB = pA + 1; pB < totalParticles; pB++){
11      ...
12      if(/** pB inside the radius of pA**/){
13        ...
14        /** Calculating thirds Newton's law */
15        fx[pB] -= tmpFx;
16        ...
17      }
18    ...
```

Figure 6.9: Particles - Code snippet of the approach that uses arrays to replicate the forces.

Unfortunately, the approaches illustrated in Figures 6.8 and 6.9 pollute the base code with data unrelated to the domain concerns. Moreover, in both approaches the programmer has to replace the original variables (*e.g.*, *fx*) by thread-related variables (*e.g.*, *fxTD*) that are only used, and meaningful, inside a specific context (*i.e.*, force calculation). Not only is the code tangling with PRC, but some of these concerns resulted in the creation of different structures to represent the same information (*e.g.*, there are two different versions of the arrays of forces). By having in the base code two variants of the same data (*i.e.*, original and the thread-related data), the programmer introduces a semantic dependency between the two and, consequently, has to update both variants accordingly. For instance, right before and after the parallel region the thread-related data and the *problematic variables* should be initialized and updated accordingly (*i.e.*, reduction), respectively. Furthermore, at least one thread should have access to the private variables of the other threads to perform the data reductions. With the approach using arrays (Figure 6.9), threads can access the *problematic* variables of each other, however, with the *ThreadLocal* approach (Figure 6.8) additional structures are needed to hold the references to the thread local data. Conclusively, one can immediately assume that, by adding the mechanism that mimics the *set/get* approach, the base code becomes harder to maintain and understand.

Underneath the *ThreadLocal* class, there is a *hashmap* that maps between threads and their private data. Hence, compared with directly accessing the original *problematic variables*, or with accessing arrays with the thread-related variables, the thread local approach adds additional instructions. These extra instructions can result in a prohibitive overhead. To reduce this overhead, the programmer might have to resort to a similar strategy to the *set/get performance* design rule. Although the use of the approach with arrays does not add the extra instructions to access a *hashmap*, *false sharing* problems might occur. Furthermore, in that approach, the thread ID had to be passed around (line 5 of Figure 6.9) so that threads could access the position in the arrays where their data was being held (*e.g.*, line 8). Moreover, this approach is less explicitly in its intentions compared with the *ThreadLocal*. In the latter, it is immediately clear that the main purpose of that structure is to provide private data to threads.

Table 6.2: MD - The number of statements needed to implement the *set/get* approach.

	AOmpLib	Java Intrusive
Number of Statements	18	40

Table 6.2 summarizes the number of statements[12] needed to replace the *problematic variables* using AOmpLib *set/get* approach and an intrusive approach similar to the one shown in Figure 6.9. More details about the number of statements are provided in Figure A.6.

[12]Those statements do not include the creation and management of threads, parallel regions and *for* constructors, they include instead the statements related with *set/get* approach.

In the MD case study, an intrusive implementation that creates, *per* thread, private copies of the *MD* and *Particles* objects is less intrusive than an implementation that mimics the *set/get* approach. Instead of individually replacing each *problematic variable*, the programmer can replace the original *MD* and *Particles* objects with private ones (line 9 of Figure 6.10). The code that creates the private data is localized within the logic that provides the parallel region (line 4 to 11) instead of being spread over the MD and Particles classes.

```
1   void cicleForcesNewtonsLaw(){
2     epot = vir = 0.0;
3
4     for(int i = 0; i < totalThreads-1; i++){
5       final MD threadPrivateMD = this.copy();
6       final Particles threadPrivateParticles = particles.copy();
7       ...
8       teamThreads.addTask(() -> {
9         threadPrivateParticles.calculate_force(threadPrivateMD, ...);
10        ...
11      }...;
12    ...
13  }
```

Figure 6.10: MD - Code snippet of the intrusive private objects approach.

```
1  void calculate_force(MD *md)
2  {
3    // create array of forces and MD copy
4    // Initialize those variables
5    ...
6    #pragma omp declare reduction(mdReduce : MD :         \
7      omp_out.epot += omp_in.epot,                        \
8      omp_out.vir += omp_in.vir,                          \
9      omp_out.interactions += omp_in.interactions)  \
10
11   #pragma omp parallel reduction (mdReduce:mdThread) reduction(+:fx,fy,fz)
12   {
13     Particles threadParticle;
14     threadParticle.fx = fx;
15     threadParticle.fy = fy;
16     threadParticle.fz = fz;
17     ...
18     #pragma omp for schedule(static,1)
19     for(int i = 0; i < totalParticles; i++)
20       forceNewtonsLaw(&mdThread, i, &threadParticle);
21     ...
22     // Updating the original variables
23   }
```

Figure 6.11: MD - Code snippet of OpenMP parallelization of the best (so far) SM layer.

In the OpenMP C implementation that uses data redundancy approach, the *MD* and *Particles* classes were translated into C structures with the same name and fields. The parallel region and static *for* were easily inserted through annotations. However, to create and reduce the private structures additional code was necessary. In OpenMP C, annotations to reduce structures and arrays were only introduced in the versions 4.0 and 4.5 of the standard, respec-

tively. Unfortunately, neither OpenMP C nor JOMP[13] offers annotations to reduce dynamic allocated arrays/structures, which is needed in our MD C implementation. Nonetheless, we can shape the code to exploit the OpenMP annotations to reduce structures and arrays (illustrated in Figure 6.11). We could create a local *MD* structure (statically allocated) and arrays of forces and use them in the OpenMP annotations (lines 6 to 11). Afterward, instead of the original *MD* structure, the local one is passed as a parameter of the *forceNewtonsLaw* method (line 20). Moreover, to avoid changing the code inside the *forceNewtonsLaw* method, we can create a local *Particles* structure (line 13) and adjust its forces fields to point to the local arrays of forces (lines 14 to 16) and the remaining fields to the pointers in the original *Particles* structure. After the parallel region, the original variables from the *MD* and *Particles* structures have to be updated.

Table 6.3 presents the total number of statements used to implement the (so far) best SM layer. In these statements are included the creation of the parallel region, static *for*, and the replication of private *MD* and *Particles* objects.

Table 6.3: MD - The number of statements needed to implement the (so far) best SM layer.

	AOmpLib	OpenMP C	Java Intrusive
Number of Statements	29	33	49

Until this last SM layer, almost all OpenMP C implementations were less complex and required fewer statements than the intrusive and aspect implementations. However, the last SM layer reduces arrays inside structures, which is a feature currently not supported by OpenMP through annotations. Hence, the creation and management of additional temporary structures to be used in the reduction annotations clauses and to be passed as parameters of the methods performing the force calculation. Furthermore, we also had to ensure that the results of those reductions were correctly transferred to the corresponding variables in the base code.

Using the AOmpLib pointcuts to add a parallel region and a *for* is less complex than creating these constructors from scratch, but more complex than using the OpenMP annotations. Nonetheless, using AOmpLib annotations to provide the same features reduces both the complexity gap to OpenMP as well as the number of statements needed. Compared with OpenMP C and intrusive approaches, AOmpLib shined the most in the creation and reduction of private objects. With AOmpLib, the user has only to provide the pointcuts stating *when* (and *which*) objects should become private, the copy constructor of these objects and, with annotations, specify the fields to be reduced. All the cumbersome logic behind the correct reduction and update of the corresponded variables is hidden, from the user, inside the AOmpLib. Moreover, in contrast to the OpenMP C and Java intrusive implementations, the creation of private objects did not require drastic modifications to the source code (*e.g.*, adding new structures). Hence, for the (so far) best SM layer, AOmpLib provided the implementation with the lowest number of statements (Table 6.3), as complex as the OpenMP C but less complex than the Java intrusive.

[13]JOMP is based on the OpenMP 2.0 standard and therefore can only reduce primitive data types.

6.2.1.2 Dealing with load balancing

To tackle the load balancing issues we performed a set of tests on the SM layer that uses the strategy with private object copies. The base code contains the design rules previously applied, except those from the *set/get* pointcut approach[14]. We tested static and dynamic distributions with a chunk size of one and a manual distribution that divides equitably, as much as possible, the iterations of the outer loop among threads. Each of these options is analyzed next.

The first iteration of the outer loop of the force calculation computes $\texttt{N}-\texttt{1}$ pairs of particles, the second $\texttt{N}-\texttt{2}$, and so on, with `N` being the total number of particles. Hence, threads that take the initial chunks of iterations have more pairs of particles to work with. A dynamic *for* distributes iterations as the threads request them, which improves load balancing. However, a dynamic *for* has a higher overhead than a static due to the task distribution synchronization. Thus, we implemented a manual static distribution that divides the outer loop into two different loops, one that performs the iterations between 0 and $\frac{\texttt{N}}{\texttt{2}}-\texttt{1}$ and another that performs the iterations between $\texttt{N}-\texttt{1}$ and $\frac{\texttt{N}}{\texttt{2}}$. The thread that executes the iteration `i` (with $\texttt{0} \leq \texttt{i} < \frac{\texttt{N}}{\texttt{2}}$ and $\texttt{N} > \texttt{1}$) will also execute the iteration $\texttt{N}-\texttt{i}-\texttt{1}$. Hence, whenever possible threads will execute chunks of $\texttt{N}-\texttt{1}$ pairs of particles[15].

To test the different distributions of the parallel *for* no additional design rule was necessary since the *for* method was already created. We have already tested the static distribution with a chunk of one during the testing of different strategies to deal with data races. Hence, in the current phase, we only tested the dynamic and the (static) manual distributions. Moreover, to limit the number of tests, we tested the dynamic *for* distribution just with its default chunk. Pointcut-wise, testing the dynamic *for* was solely a matter of renaming the pointcut *for_static* to *for_dynamic*. To test our manual scheduling we used the user-defined *for* distribution constructor of the AOmpLib (similar to the one shown in Figure 5.27).

Figure 6.12, and Tables B.5 (execution time) and B.6 (speedups) show that the dynamic scheduling achieved the best speedups (*i.e.*, 16.45× overall). Our manual distribution suffers from load balancing problems because of the two different levels of task granularity inside the force calculation – when particles are within the same radius and when they are not. In the former, extra logic is performed to calculate the force between the two particles, which results in a higher time spent to complete that task than when the particles are not in the same radius.

[14]Code readability is the reason why these design rules were excluded.

[15]The iteration `i` has $\texttt{N}-\texttt{1}-\texttt{i}$ pairs of particles and the iteration $\texttt{N}-\texttt{i}-\texttt{1}$ has $\texttt{N}-\texttt{1}-(\texttt{N}-\texttt{i}-\texttt{1})$ pairs of particles. Thus, $(\texttt{N}-\texttt{1}-\texttt{i})+(\texttt{N}-\texttt{1}-(\texttt{N}-\texttt{i}-\texttt{1}))$ which simplifying is $\texttt{N}-\texttt{1}$. The exception to this is the iteration $\frac{\texttt{N}}{\texttt{2}}-\texttt{1}$ when `N` is an even number (*e.g.*, fourth row of Figure 6.4). For that iteration threads will calculate $\frac{\texttt{N}}{\texttt{2}}$ pairs of particles instead.

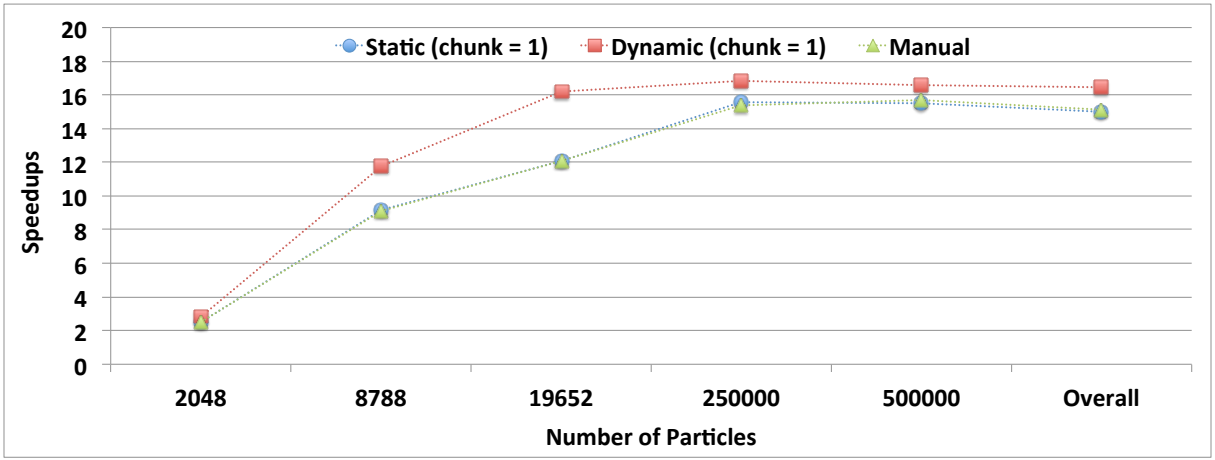


Figure 6.12: MD - Speedups of the strategies to deal with load balancing in SM.

Testing different *for* distributions with AOmpLib or OpenMP required practically the same effort and reasoning. Except for the manual, testing *for* distributions required changing only the name of the pointcut/annotation. Because OpenMP does not provide customized distribution through annotations, the user has to manually change the *for* body directly in the base code, resorting to explicit calls to methods of the OpenMP API (*e.g.*, getting the total number of threads and their ids). With AOmpLib or OpenMP, the user performs a similar effort, however, in AOmpLib the logic is coded in the layer instead of directly in the base code. Naturally, testing different *for* distributions with the intrusive implementations requires more effort than with AOmpLib or OpenMP because the programmer has to rewrite the loop from scratch and guarantee that the heuristic of the distribution is correct. Such a task can be cumbersome, especially with chunk sizes greater than one, which typically results in adding an extra loop to iterate over the iteration chunks. To reduce future work the user can save the heuristics of loop distributions in utility-type classes to reuse that knowledge in different applications. Nevertheless, even in that case, the user still needs to rewrite the *for*.

After the SM testing phase, we ended up with parallelizations (*i.e.*, aspects, OpenMP C, and Java intrusive) that, for each thread, create private copies of the *MD* and *Particles* objects and that dynamically distribute, among threads, the force calculation between a particle and the remaining. Moreover, at the end of the parallel region, the *epot*, *vir*, and *interactions* field variables of the *MD* and the arrays of forces of the *Particles* objects are reduced.

6.2.2 Distributed Memory

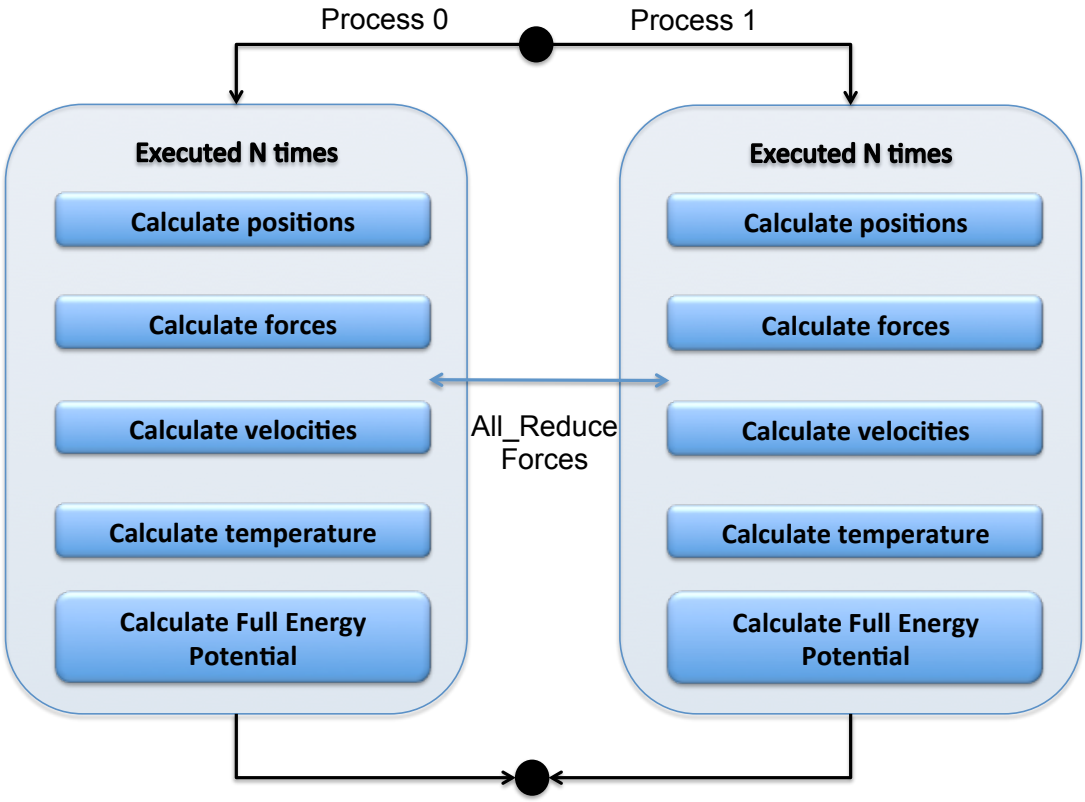


Figure 6.13: MD - DM parallelization with two processes.

The initial DM layer used in this subsection and illustrated in Figure 6.13 with two processes is the one shown previously in Section 5.3.1 (*i.e.*, Figure 5.12). In this layer, each process runs the entire MD simulation, but during the force calculation, each process is only responsible for calculating the forces of a subset of particles. After completing the force calculation, processes exchange the particles forces among them using the MPI *Allreduce* routine. Alternatively, processes could send only the forces that they updated, but such an approach would require to keep track of the forces to be exchanged.

We could have developed a DM layer similar to our SM layer, where only the master process would execute the entire simulation, while the others would only perform the force calculation. However, since before the force calculation, the updated positions of the particles are required, the processes would have to broadcast the particles' position at that time. Consequently, this approach would have added an extra communication overhead compared with the current one. The current approach removes the need to send the particle's position to processes, before the force calculation, since the processes compute these positions locally. Hence, reducing the amount of communication needed.

As with the tests performed for the SM layer, for the DM version we also tested three different *for* scheduling approaches, namely, manual, static and dynamic, the last two with a chunk of one. The manual scheduling employs the same heuristic as the one used in SM. The results of these tests are shown in Figure 6.14 and in Tables B.5 (execution time) and B.6 (speedups). In the DM tests, as with the SM tests, the static and manual distributions achieved similar speedups, with the former achieving the highest overall speedup of 15.68× while the later 15.67×. However, contrary to what happened in the SM, the DM dynamic for had overall the worst performance (15.52×), possibly because the communication used to dynamically distribute the tasks produced a higher overhead than the lock mechanism used in the SM dynamic loop distribution. In the best SM layer, the dynamic distribution achieved the best speedups for all the inputs, whereas in the best DM layer only for the two largest inputs. Compared with the best SM layer, the best DM layer scales more for the first three inputs and slightly less for the remaining.

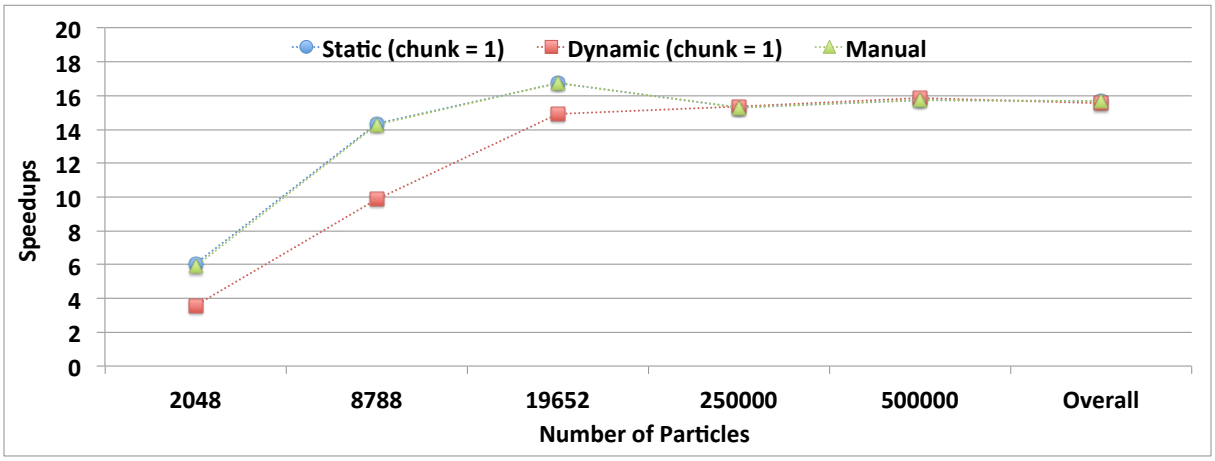


Figure 6.14: MD - Speedups of the strategies to deal with load balancing in DM.

Table 6.4 presents the number of statements needed to implement the best DM layer using the AOdmLib and a Java intrusive approach.

Table 6.4: MD - The number of statements needed to implement the DM Layer.

	AOdmLib	C MPI	Java MPI Intrusive
Number of Statements	21*	20	31

* With annotation that value is only 10.

6.2.3 Hybrids: Composing the Best SM and DM Layers

With our libraries of aspects, building the hybrid version was as straightforward as adding the best SM and DM layers into the build. A possible source of issues during the composition of the two aspect layers could have been the interception of the same join points by both – the only occurrence of such join point is the method on which the *for* design rule was applied (*i.e.*, the *calculate_force* method call). From the SM layer, this join point is intercepted by four pointcuts, namely those providing the parallel region, dynamic *for* distribution and the two related with the creation of private objects, whereas from the DM layer is intercepted by the pointcut providing the static *for* distribution. Based on the precedence rules of our framework, this join point is intercepted first by the parallel region pointcut and then by the pointcuts concerning the creation of private objects. Afterward, between the *for* distribution pointcuts, the one from the DM layer is injected first and only then is the one from the SM layer injected as well. Although the correct composition of the SM and DM layers could have been affected by the order on which these code transformations occurred (over the *calculate_force* method), the framework ensured its correctness.

Building the OpenMP + MPI hybrid version was almost the same as merging the two codes. In the Java intrusive hybrid version, it was necessary to adapt the dynamic *for* distribution from the SM code to also take into account the number of MPI processes.

6.2.4 Performance evaluation

To evaluate the overhead of our approach we measured the execution time of:

- the sequential (base) code with and without the design rules;
- the SM and DM intrusive versions *versus* AOmpLib and AOdmLib, respectively.

We also compared these versions to their corresponding C implementations.

Regarding the results of the sequential versions, our design rules added no overhead to the original base code and the execution time of the C implementation was, in total for all the input sizes, approximately equal to the Java implementation. For the three smallest input sizes, the C implementation was from 1.01 to 1.05× faster than the Java implementation, but identical for the two largest (these results are detailed in Tables B.7 and B.8).

In the MD case study, the design rules that could have lead to a noticeable overhead would have been those introduced inside the method of the force calculation since this method takes 99% of the simulation execution time. However, all of these design rules are methods that ended up being inlined by the JVM[16].

[16]This was confirmed by using the JVM flags *-XX:+UnlockDiagnosticVMOptions* and *-XX:+PrintInlining*.

Figures 6.15 and 6.16 show the results of the Java/C implementations that mimic the best SM and DM layers. More details about these results are provided in Tables B.9 (execution time), B.10 (speedup), and B.11 (gains).

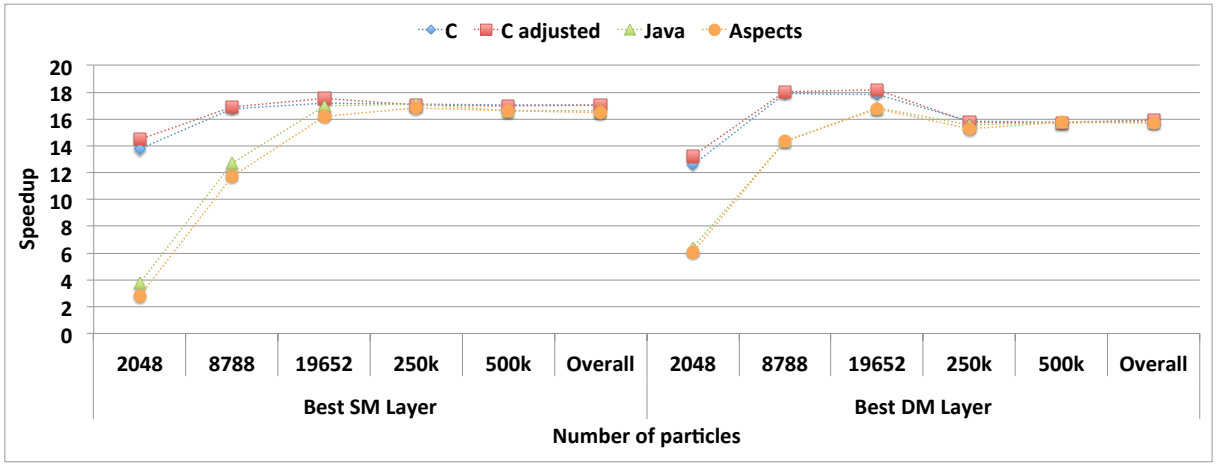


Figure 6.15: MD - Speedups of the SM/DM versions.

In Figure 6.15, the difference between *C* and *C adjusted* is that in the former the speedups are calculated based on the C sequential execution time, whereas in the latter are based on the Java sequential execution time. The curves of C and C adjusted overlap each other because the execution time of the C and Java sequential versions is practically the same. With our SM and DM aspect layers, we scaled the MD case study to an overall speedup of 16.45× and 15.68×, respectively – 16.62× and 15.77× with the Java intrusive implementations. Unfortunately, for the two smallest inputs the speedups of the Java and aspect implementations were under 15×, and consequently, these inputs were not used in our hybrid versions.

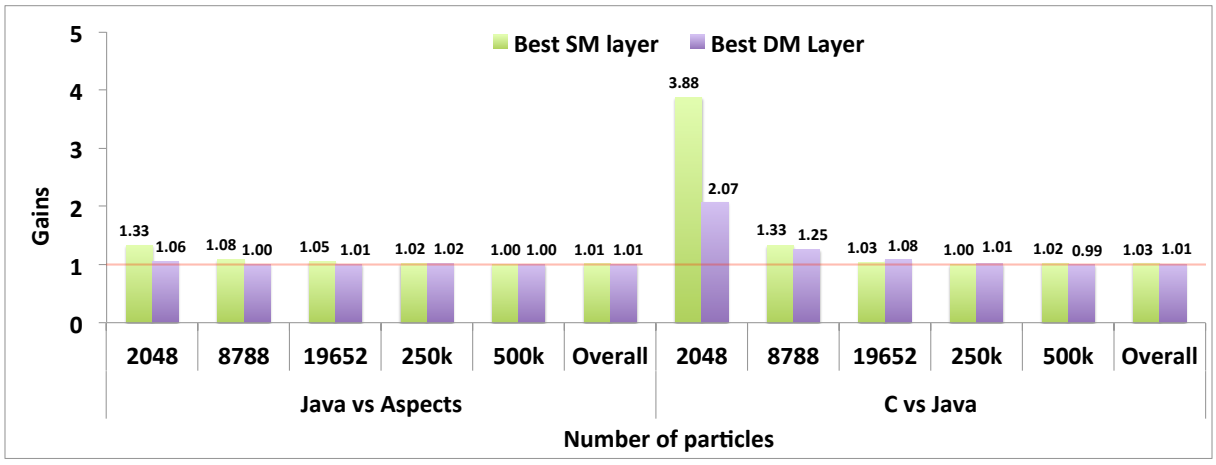


Figure 6.16: MD - Comparing parallel implementations in SM and DM environments.

Figure 6.16 shows that the performance differences between Java and aspects (*i.e.*, *Java vs Aspects*) and between C and Java (*i.e.*, *C vs Java*) are higher with the smaller inputs and decreases with the increase of the number of particles. Nonetheless, the difference in absolute time between implementations was insignificant, however, percentage-wise the difference is more noticeable because the smaller the input, the shorter the execution time. For all the inputs, in both SM and DM, the time difference between the Java intrusive and aspect implementations is under 0.25 seconds, and for the C and Java is less than 1.03 seconds. Lastly, the SM and DM Java intrusive implementations, in overall, were 1.01× faster than the aspects.

Since the best SM and DM versions scaled over 15× (for the three largest sizes), we built hybrid versions out of them. To understand the benefits of the hybrids we tested and compared them with versions that only use processes (DM) in a cluster of 8 machines. In the tests of the hybrid versions, we assigned one process *per* machine, each with multiple threads, whereas for the DM tests we assigned multiple processes *per* machine. The most relevant results of these tests are shown in Figures 6.17, 6.18 and 6.19. Additionally, for the DM and hybrid versions, Tables B.12, B.13, and B.14 present information about the execution times, speedups, and gains, respectively, and Table B.15 shows the gains of the hybrids *versus* DM.

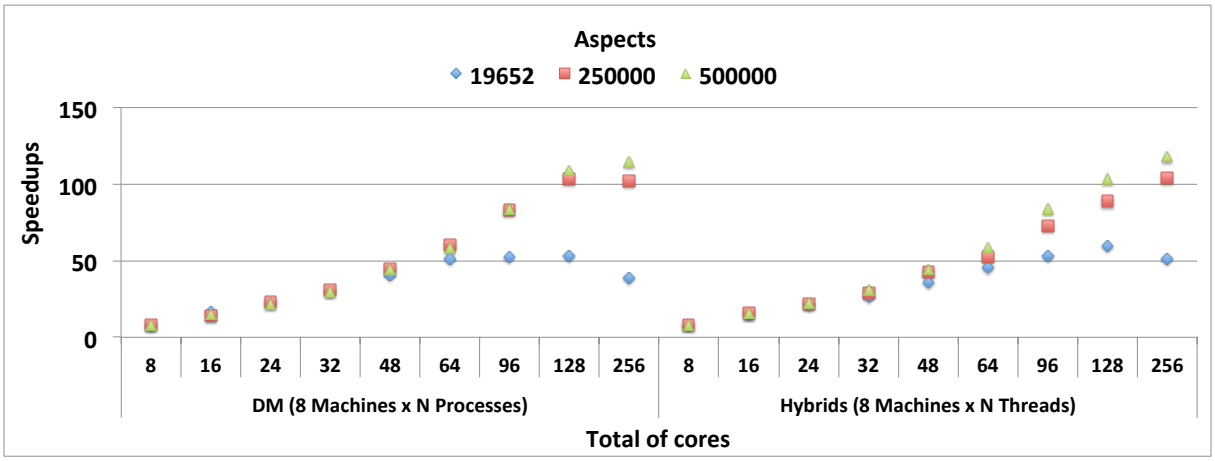


Figure 6.17: MD - Scalability of the DM and Hybrid versions in 8 machines.

Comparing the results of the layer of aspects shown in Figure 6.15 with those presented in Figure 6.17, we can see that, regardless of the version (*i.e.*, DM or Hybrid), the speedups achieved with multi-machines surpassed those with a single machine. The other DM/Hybrid implementations (C and Java)[17] have similar tendencies to the ones shown in Figure 6.17. By comparing the input with 19652 particles with the others, we can see that, for 19652 particles, the scalability of the DM and hybrid versions starts to decrease slightly at 48 processing units and stops at 128. Moreover, we can see that at 96 processing units the decline is more accentuated in the DM version than in the hybrid version – this is the point where communication overhead

[17]The charts of all versions are shown in Figure B.1.

starts to become more noticeable. Firstly because at this point, in the DM version, processes have to switch from communicating with Myrinet[18] to Ethernet and secondly because the overhead of communication among the 96 processes starts to outweigh the gains from the additional number of smaller parallel tasks. Nonetheless, it is worth recalling that the force calculation has a computational quadratic asymptotic complexity, which means that with the doubling of the input size the work to be computed quadruples, which helps to explain why the larger input sizes continue to scale reasonably well even with more than 96 processing units. The larger the size, the more parallel work there is, which in turn improves the load balancing. Furthermore, an increase in input size leads to an increase in the ratio of computation to communication.

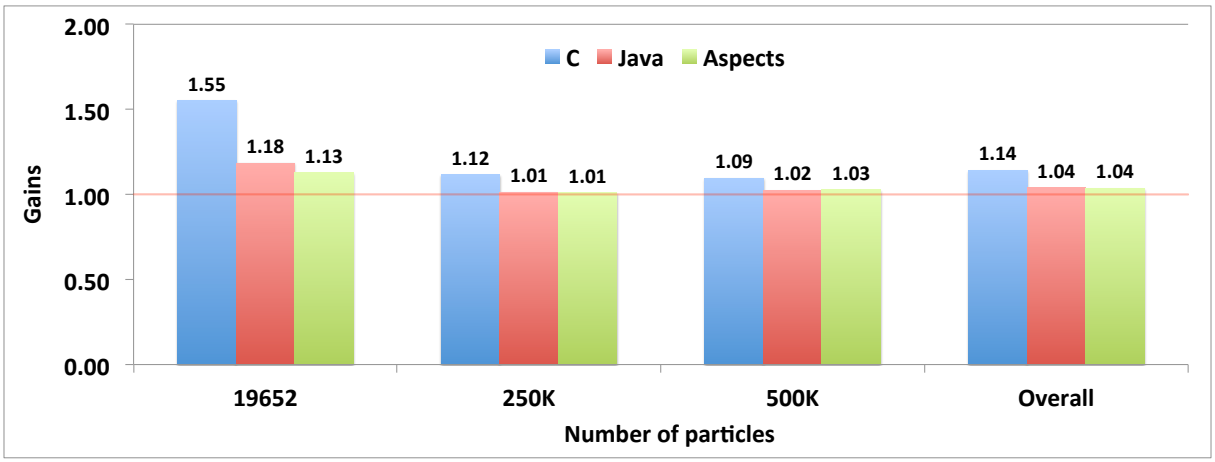


Figure 6.18: MD - Hybrids *vs.* DM versions running in 8 machines.

Figure 6.18 presents the gains (in a cluster of 8 machines) of the hybrids *versus* the DM versions implemented with C, Java, and our libraries of aspects. Regardless of the implementation and input size, the hybrid versions were faster than the DM versions. One of the reasons for that is the lower communication overhead of the hybrid versions. The communication overhead of the DM version was higher due to the use of more processes (*i.e.*, 256 instead of 8) and the use of Ethernet instead of Myrinet. Although the difference between the communication overhead of the hybrid and DM versions affects their scalability, additional factors mitigated the impact of that overhead. For instance, both versions perform communication among the same number of machines – inter-machine communication has a higher overhead than intra-machine communication. Moreover, the communication directive used (*i.e.*, `MPI_Allreduce`) has a complexity of $O(\log_2 P)$, where P is the total number of processes.

In the MD case study, the hybrids compared with the DM versions, have the advantages of using less memory (*i.e.*, fewer JVMs) and exploiting more efficiently the dynamic scheduling. However, the hybrids require an extra reduction step (among threads) and the creation of thread private data. In the Java and aspect implementations, for 19652 particles, the hybrids surpass

[18]Our Myrinet was limited to 8 ports.

the DM at 96 processing units, whereas for the remaining inputs just during hyper-threading (*i.e.*, 256 processing units). However, in C, for all input sizes, the hybrid surpass the DM at 48 processing units. This result may indicate that the hybrid versions implemented with Java or aspects have a higher overhead than these implemented with C. Consequently, explain why the highest overall gain of the hybrids over the DM versions was in the C implementations – 1.14× compare with 1.04× of the Java and aspect implementations.

Figure 6.19 confirms our previous suspicion since with the hybrids C was 1.15× faster than Java, whereas with the DM it was "only" 1.05×. These results imply that the SM part of the hybrid versions implemented with either Java or aspects has a considerable overhead comparing with C. Regarding Java *versus* Aspects, overall in the DM and hybrid versions, Java was 1.01 and 1.02× faster than our libraries of aspects, respectively.

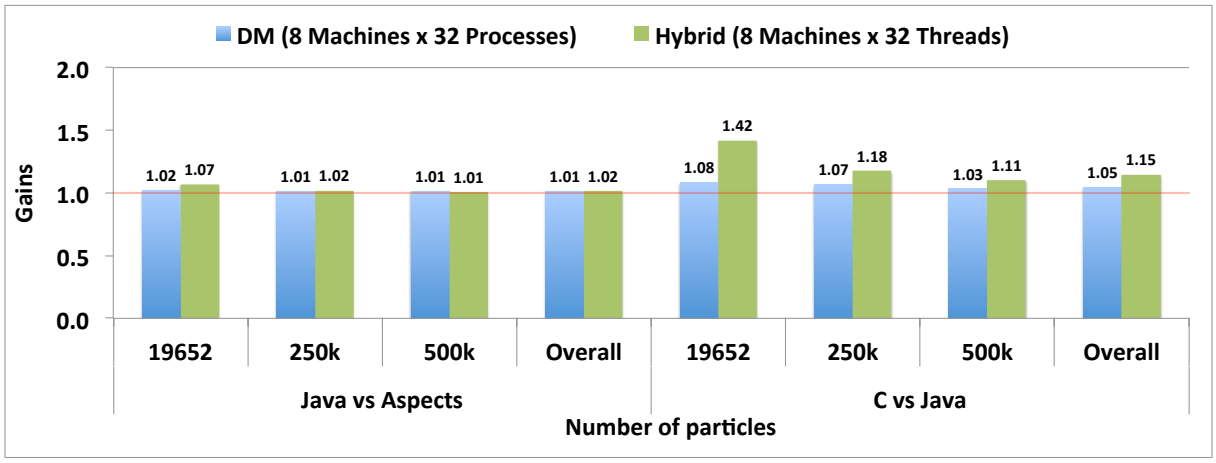


Figure 6.19: MD - Comparing parallel versions in DM/Hybrid with 8 machines.

After profiling, we found out that one of the sources of overhead in the hybrids implemented with Java and aspects came from the creation of the thread private *MD* and *Particles* objects in parallel. We rewrote the Java hybrid version[19] to instruct the master thread to create and assign the objects to the corresponding threads in a sequential manner. We tested again to verify the differences between the old and new Java hybrid versions, the results of these tests are presented in Tables B.16 and B.17. The scalability trend of this new hybrid is similar to the one described for the C implementation. This new Java implementation is overall 1.10× faster than the aspects (0.8 more than the old one) corresponding to an overall reduction of 0.86 seconds in execution time (0.7 more than the old implementation). The overall gain of C over Java was reduced from 1.15 to 1.07×, which corresponds to reducing the difference between execution times of C over Java from 1.24 to 0.55 seconds.

[19]We did not update the C code because it did not create objects in parallel, in fact, the new Java implementation is more similar to the C code than the old Java implementation.

6.3 Case Study : Matrix Multiplication

Our matrix multiplication (MM), exemplified in Figure 6.20, instead of performing an elementary multiplication element by element of the matrix C, uses an efficient tiling approach based on [SGS+14] specially tuned for Java. The MM is sub-divided into the multiplication of smaller matrices (tiles). Our algorithm was thoroughly developed to further sub-divide those smaller matrices into even smaller ones to fully exploit the three levels of cache.

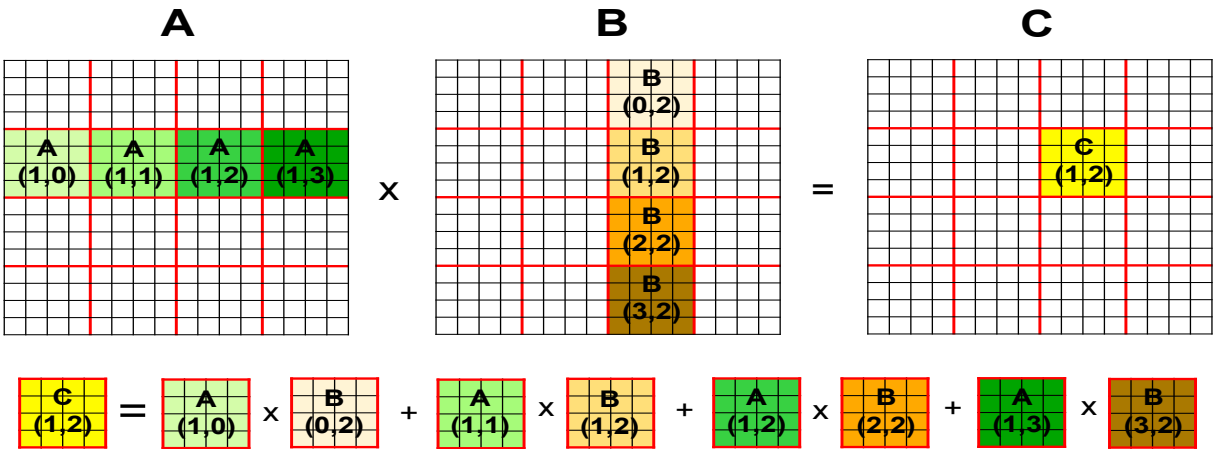


Figure 6.20: MM - Tiling illustration.

Figure 6.21 presents part of the code from our sequential MM version. The MM of a given matrix C is divided into the MM of smaller tiles[20], in our case with size `32` × `256`. The *matrixMultiplication* method of Figure 6.21 presents the structure of the MM kernel composed by outer and inner loops (lines 23 and 25) that goes through the titles horizontally and vertically, respectively. For each `jj` iteration of the outer loop, our MM loads a chunk of 256 consecutive columns of the matrix B into the L3 cache, which is shared among cores. These columns of the matrix B are loaded, inside the *packingCacheL3* method, into a sub-matrix labeled `bb`. The transfer of values from the columns of matrix B is done by loading 4 lines at a time and placing them in a single line of matrix `bb`[21] to make the MM more cache-friendly. Afterward, for each `ii` iteration of the inner loop (line 25), a chunk of 32 consecutive lines of the matrix A are loaded and used inside the *microTiling* (line 27) method. The *microTiling* method performs the MM of the tile `ii` × `jj` from matrix C by multiplying the lines loaded from matrix A with the columns of matrix B stored in the sub-matrix `bb`. The *microTiling* method further splits the MM of the tile `ii` × `jj` into smaller chunks to take advantage of the layout of the matrix `bb`, the smaller cache levels (*i.e.*, L1 and L2), and the registers. For brevity, the code of the *microTiling* method was omitted since it is not fundamental for the understanding of the MM parallel versions.

[20] We found out that tiles with size `32` × `256` provide the best execution times for the sequential MM. Consequently, and to reduce the number of possible future tests, we used tiles with size `32` × `256` for the parallel versions of the MM, as well.

[21] Detailed code and visually illustration in Figures D.1 and D.3 of Appendix D, respectively.

```
1  public final class MM {
2
3    public final static int tilei = 32, tilej = 256; // Tile size
4    private final double A[][], B[][], C[][];
5    private final int maxRowA, maxColA, maxRowB, maxColB ...;
6
7    public MM(int mRA, int mCA ,int mRB, int mCB) {
8      maxColA = mCA; maxRowA = mRA;
9      ...
10     A = new double [maxRowA][maxColA];
11     ...
12   }
13   ...
14   public void packingCacheL3(int jj, double bb[][]){
15     for(int k = 0; k < maxRowB; k += 4)// Loads 4 lines
16       ...
17   }
18   ...
19   public void matrixMultiplication(){
20     final double bb[][] = new double[maxRowB/4][tilej*4+1]
21     final double cc[][] = new double[tilei][tilej+1];
22
23     for(int jj=0; jj< maxColC; jj += tilej) {
24       packingCacheL3(jj, bb);
25       for(int ii = 0; ii< maxRowC; ii += tilei){
26         initBlock(cc);
27         microTiling(ii, bb, cc);
28         updateCCmatrix(ii, jj, cc);
29       }
30     }
31   }
32 ...
33 }
```

Figure 6.21: MM - Relevant code of the sequential version.

It is possible to identify (at least) two potential sources of parallelism in the *matrixMultiplication* method, namely the iterations of the outer and inner loops that can be assigned to threads/processes. There are different trade-offs between parallelizing the outer and the inner loops; for instance, the former has fewer iterations but more computation *per* iteration than the latter. In the one hand, parallelizing the outer loop might lead to load balancing problems due to having a small number of iterations. On the other hand, the iterations from the inner loop might not have enough granularity to overcome the parallelization overhead. Since our MM uses tiles with size 32×256, the inner loop has 8 times[22] more iterations that the outer loop. Furthermore, in the SM version, parallelizing the inner loop allows sharing the sub-matrix `bb`[23] across threads, in contrast with parallelizing the outer loop which requires threads to have their private `bb` sub-matrix. Regarding the DM versions, the processes to correctly perform the MM of the tiles assigned to them only need specific chunks of the matrices. Hence, we can use an approach that splits the matrices among processes. In practice, this approach works the same as distributing the loop iterations among processes, but without having to waste memory in redundant data. As illustrated in Figure 6.20, a given $\texttt{Tile}_{\texttt{ii}\times\texttt{jj}}$ of size $\texttt{tilei} \times \texttt{tilej}$ from

[22] tilej/tilei = 256/32 = 8.

[23] According to our MM implementation, the sub-matrix `bb` is loaded into the shared L3 cache.

the matrix C is the result of multiplying a complete line of $\texttt{Tiles}_{\texttt{ii}\times\alpha}$ from the matrix A with $\texttt{0} \leq \alpha < \frac{\texttt{maxColA}}{\texttt{tilei}}$ by a complete column of $\texttt{Tiles}_{\beta\times\texttt{jj}}$ from the matrix B with $\texttt{0} \leq \beta < \frac{\texttt{maxRowsB}}{\texttt{tilej}}$. Thus, there are two main DM versions of this MM implementation:

1. assigning lines of C tiles to processes, each process needs the entire matrix B and the appropriate chunks of lines from the matrices A and C;

2. assigning columns of C tiles to processes, each process needs the entire matrix A and the appropriate chunks of columns from the matrices B and C.

Analogously, the first and second approaches correspond, in the SM version, to distribute the outer and inner loop iterations among threads, respectively.

Similar to what was done in the MD case study, we tested different strategies of loop/matrices distribution in the SM/DM versions with aspects. The best MM SM layer found creates a parallel region around the *matrixMultiplication* method and dynamically distributes the loop iterations of the *packingCacheL3* method and the inner loop iterations of the *matrixMultiplication* method. Additionally, the sub-matrix `bb` is made shared among threads, whereas the `cc` remains private. Our SM version needs synchronization barriers after the *packingCacheL3* method and in-between the outer loop iterations of the *matrixMultiplication* method. The first barrier ensures that the shared sub-matrix `bb` has all its values loaded before threads can work with them in the MM tiling, while the second guarantees that threads will not execute the *packingCacheL3* method before the others have finished the micro MM tiling of their tiles. Since both the *packingCacheL3* method and the inner loop use the *for* constructor, they have already implicit barriers at the end of their execution. Moreover, there is no risk of deadlock, since all threads execute all the iterations of the outer loop, from the *matrixMultiplication* method, hence every thread will reach all the barrier calls within the scope of that loop.

Our best DM version splits the matrices A and C by chunks of lines (of size `tilei`) across processes while replicating the entire matrix B. In that version, the master process is responsible for scattering, broadcasting, and gathering the matrices A, B, and C across/from the remaining processes, respectively. Naturally, the scattering and broadcasting occur before the *matrixMultiplication* method and the gathering after it. Regarding the hybrid version, besides testing the composition of the best SM and DM layers, we also tested the use of the DM layer that splits the matrices B and C by chunks of columns (of size `tilej`) across processes while replicating the matrix A, combined with the best SM version. This alternative hybrid approach conceptually corresponds to statically distributing the outer loop iterations of the *matrixMultiplication* method among processes combined with dynamically distributing both the inner loop iterations of the *matrixMultiplication* method and the outer loop iterations of the *packingCacheL3* method among the threads of each process. To test that hybrid version we reused both the layer created for the best SM version and one of the DM layers created during the intermediate DM tests. In the end, the best hybrid version remained the one that combines the best SM and DM versions.

6.3.1 Programmability evaluation

Our best SM layer uses one parallel region, a *single* and two dynamic *fors*, all of them implicitly call a barrier. To introduce these parallel constructors three design rules were applied, one to intercept object creation and two *for* methods. The first design rule was used to provide a shared `bb` memory reference among threads[24], whereas the last two were necessary to inject the *for* constructors. Regarding the parallel region, no extra design rule was needed since this constructor is applied over a method that already existed in the base code (*matrixMultiplication*).

```
1  public double[][] createPackBB() {
2    return new double[maxRowB/4][tilej*4+1];
3  }
4  public void mmTitleMatrixC(int begin, int end, int step, ...){
5    for(int ii = begin; ii< end; ii += step){
6      initBlock(cc);
7      microTiling(ii, bb, cc);
8      updateCCmatrix(ii, jj, cc);
9    }
10 }
11 public void matrixMultiplication(){
12   final double bb[][] = createPackBB();
13   final double cc[][] = new double[tilei][tilej+1];
14
15   for(int jj=0; jj< maxColC; jj += tilej){
16     packingCacheL3(0, maxRowB, 4, jj, bb);
17     mmTitleMatrixC(0, maxRowC, tilei, bb, cc, jj);
18   }
19 }
```

Figure 6.22: MM - Application of the SM design rules.

One of the *for* design rules (shown in Figure D.2) simply reused the *packingCacheL3* method, but adapting its signature, call, and outer loop range accordingly. For the remaining design rules, two new methods were created, namely *createPackBB* (line 1 of Figure 6.22) and *mmTitleMatrixC* (line 4) to apply the *single* and *for* constructors, respectively. The changes in lines 12, 16 and 17 of Figure 6.22 resulted from applying the *object creation* design rule, the injection of the loop range from the *packingCacheL3* method and the call to the *for* method that encloses the old inner loop, respectively.

Figure 6.23 shows the SM layer of aspects using annotations (the equivalent pointcut version can be seen in Figure D.5). The parallel region, the *single*, and the two dynamic *fors* are presented in the lines 3, 4, 5, and 6 of Figure 6.23, respectively.

```
1  public final aspect SM_MM extends SM_Layer {
2      ...
3      declare @method: * *.MM.matrixMultiplication()   : @Parallel;
4      declare @method: * *.MM.createPackBB()           : @Single;
5      declare @method: * *.MM.mmTitleMatrixC(..)       : @For_dynamic;
6      declare @method: * *.MM.packingCacheL3(..)       : @For_dynamic;
7  }
```

Figure 6.23: MM - The best SM layer using annotations.

[24] Applying a *single* over an *object creation* method will provide a shared memory reference among threads.

To replicate the best SM layer with a Java intrusive implementation, we started by creating a parallel region (exemplified in Figure D.4). Creating a simple parallel region, even with the use of lambdas, required almost the same amount of code used to create the entire SM layer with pointcuts and nearly twice as much as with annotations. To build the parallel region we created a new method that wraps around all the logic to implement it. Inside the new method, the parallel tasks are assigned to a pool of threads, and the sub-matrix `bb` is created before the task assignment and passed as the argument of the new method so that it is shared among the threads inside the parallel region.

```
1  ... AtomicInteger tasksPackingLoop = new AtomicInteger(0);
2  ... AtomicInteger innerLoop = new AtomicInteger(0);
3  ...
4  public void matrixMultiplication(int threadID, double [][] bb){
5    ...
6    for(int jj=0; jj< maxColC; jj += tilej){
7      ...
8      if(threadID == 0) innerLoop.set(0);
9      callBarrier();
10     for(int ii=innerLoop.getAndAdd(tilei); ii< maxRowC; ii=innerLoop.getAndAdd(tilei)){...}
11
12     if(threadID == 0) tasksPackingLoop.set(0);
13     callBarrier();
14   }
15 }
```

Figure 6.24: MM - Dynamic intrusive *fors* in the SM Java intrusive version.

After creating the parallel region, we implemented the two dynamic *fors*. To simulate the counters of these *fors* we used variables with atomic properties (lines 1 and 2 of Figure 6.24) that are initialized (by the master thread) between iterations of the outer loop (lines 8 and 12). Afterward, we adapted the appropriate loops to use the dynamic counters (*e.g.*, line 10).

Table 6.5 shows the total of statements needed to implement the best SM layer (more detail in Figure D.6). The annotation-based approaches (AOmpLib, OpenMP, and JOMP) required the least number of added/modified statements, followed by the pointcut-based and the Java intrusive approaches. Regarding complexity, naturally, the intrusive approach was the most complex followed by the pointcut approach and with the annotations being the least complex. The hardest part of the Java intrusive approach was to correctly introduce and initialize the variables used in the dynamic *for* distribution (making sure it was free of data races). From the AOmpLib side, all the necessary design rules were applied automatically using the *method refactoring* and *introduce parameter* features from eclipse. Replicating the best SM layer with OpenMP C and JOMP was only a matter of adding three annotations to the right code spots. Unfortunately, with JOMP it was also necessary, due to its restrictions, the removal of the *final* clause of some variables (*e.g.*, sub-matrix `bb`). The noticeable difference between the number of statements used with AOmpLib and OpenMP annotations come from the fact that we also included the statements required for the design rules (*i.e.*, 8 statements). The AOmpLib pointcut and annotations based approaches *per se* only required 8 and 5 statements, respectively.

Table 6.5: MM - The number of statements needed to implement the SM Layer.

	AOmpLib	OpenMP	JOMP	Java Intrusive
Number of Statements	16*	3	10	33

* With annotations the number of statements is 13.

Our best DM version uses 5 aspects, 2 to split the matrices A and C by lines among processes (data partitioning feature), one to broadcast the matrix B across processes and another to allow the execution of certain methods by the master process only. In the data partitioning aspects, we used pointcuts to send the dimensions of the matrices A and C that each process should allocate, to have a global view over the matrix A when populating it, and to send to each process the values of their corresponding sub-matrix A. To broadcast the matrix B we used an *all reduce*[25]. Finally, we made sure that the master process is the only one allowed to populate the matrices A and B and to validate the matrix C.

```
1  public final class MM {
2    ...
3    public MM(int mRA, int mCA ,int mRB, int mCB){
4      A = createMatrixA(mRA, mCA);
5      ...
6      maxColA = (A.length > 0) ? A[0].length : 0;
7      maxRowA = A.length;
8      ...
9    }
10   ...
```

Figure 6.25: MM - The DM design rules in the MM object constructor.

The best DM layer required 3 new design rules along with some refactoring resulting from collateral side effects of the DM model itself. Line 4 of Figure 6.25 shows the application of *object creation* design rule over the allocation of the matrix A (we omitted the code related with the other matrices). These design rules provided the AOdmLib with hook points to inject the local/global matrix views of the data partitioning feature.

Comparing the code of Figure 6.25 with the base code (lines 7 to 12 of Figure 6.21), besides the introduction of the design rules, we can see that the assignments to the variables holding the sizes of the matrices along with their positions in the code have changed. These modifications were necessary because, unlike with the sequential version, with the DM versions matrices might end up with different sizes than the ones initially defined. Furthermore, because some processes might end up without sub-matrices, additional safety checks were necessary (line 6 of Figure 6.25). The order of the assignments to the variables holding the sizes of the matrices also changed, because the matrices have to be created before we can query them for their sizes. Moreover, because the memory reference of the matrices may vary at runtime, due to the local/global view feature from AOdmLib, the *final* clauses from the matrices fields' declaration

[25] In theory, the function most suitable to this task should have been an actual MPI broadcast. However, in the MPI version that we used the broadcast function was, in overall, considerably slower than the *all reduce*. We also tested the broadcast algorithm with tailored-made MPI flags, but overall the *all reduce* function remained the fastest.

had to be removed. Although, it was not necessary in the MM case study a more robust solution would have been the removal of the fields that statically hold the sizes of the matrices and the replacement of their occurrences by querying the matrices themselves for their sizes.

Table 6.6: MM - The number of statements needed to implement the DM Layer.

	AOdmLib	Java	C
Number of Statements	38	51	38

During the implementation of the best DM layer with Java/C MPI we experienced the same collateral side effects encountered in the implementation with AOdmLib. Table 6.6 presents the number of statements needed to develop the best DM layer (more detail in Figure D.7).

Regarding complexity, the C/Java intrusive implementations were more complex than the AOdmLib, since the programmer must code all the logic to scatter/gather the matrices among processes in chunks of size `tilei`. Moreover, coding the intrusive versions was not as complex as it could have been, because we had already tested and found the best layer with the AOdmLib. Therefore, not only we did not have to code all the different approaches to scatter/gather the matrices (*e.g.*, by lines or columns), but also when we coded the intrusive versions we already knew the correct heuristics to use, which made the task less cumbersome and error-prone.

The best hybrid version is the composition of the best SM and DM layers. Summarizing, in the best hybrid version each process performs the MM of a subset of lines of the matrix C, which in turn is divided into smaller ones that are dynamically assigned to the threads created in the scope of the process. The composition of the best SM and DM layers of aspects was performed seamlessly, without requiring any modifications or additional design rules. The call to the *matrixMultiplication* method was the only join point that had multiple interceptions by the two layers. Hence, that join point could have led to the incorrect composition of the layers. From the DM layer, that join point is intercepted by three pointcuts, namely the ones providing the scattering of the matrix A, the gathering of the matrix C and the reduction of the matrix B. Additionally, from the SM layer, that join point is intercepted by the pointcut providing the parallel region. According to the chosen pointcuts, ours and the AspectJ precedence rules, the scatters and the reduction will occur before the parallel region and the gathers after. By default, the reduction would occur after a given join point, but in this case, happens before the parallel region because we explicitly choose the pointcut (*before_comm*) that injects behavior before a join point since the processes need the values of the matrix B before performing the MM kernel.

The C/Java hybrids were created by intrusively merging the code from the SM and DM intrusive versions. Regarding the parallel loops, we took those from the SM modules without any adaptation. In contrast to the SM version, the DM version does not change the range of the parallel loops directly, but rather the size of the matrices. Hence, unlike the parallel loop from the Java hybrid version of the MD, that we had to adapt because the SM and DM versions modified the range of that loop, in the MM case study no adaption is necessary.

6.3.2 Performance evaluation

Before analyzing the results, we provide first some relevant context from the MM versions/tests. Concerning the C implementation, it is noteworthy that its versions use matrices allocated in continuous memory positions. Hence, the C versions are more cache friendly and, potentially, have a lower communication overhead in DM compared with the other versions. In the DM layer of aspects, each process has the entire matrix B and sub-matrices of the matrices A and C, additionally, the master process also keeps the entire matrices A and C. For the DM Java/C intrusive versions we applied a small optimization where the master process only works with the global matrices and does not have any sub-matrices. Finally, because of memory limitations, for the MM of matrices with `16384x16384` (the largest input), the DM tests were limited to a maximum of 16 processes *per* machine.

The input sizes used in the MM (presented in Table C.1 from Appendix C) are considerably larger than those used in the MD, with the biggest one reaching up a little bit over 6 GB. The sequential code of our MM was also tuned to a set of JVM flags[26] to improve its performance. These flags were also used in our parallel versions. Table C.3 presents the execution time of the sequential versions with and without the JVM flags.

Figure 6.26 shows the comparison of different sequential, SM and DM versions. Aside from the C implementations, the execution time of the remaining tests is the best between running the test with and without flags. For the SM and DM versions of the C *versus* Java, we removed the values of the smallest input because they were too big when compared with the remaining values, which compromised the legibility of the chart. Nonetheless, the execution times and gains of the SM and DM versions are detailed in Tables C.5 and C.7, respectively.

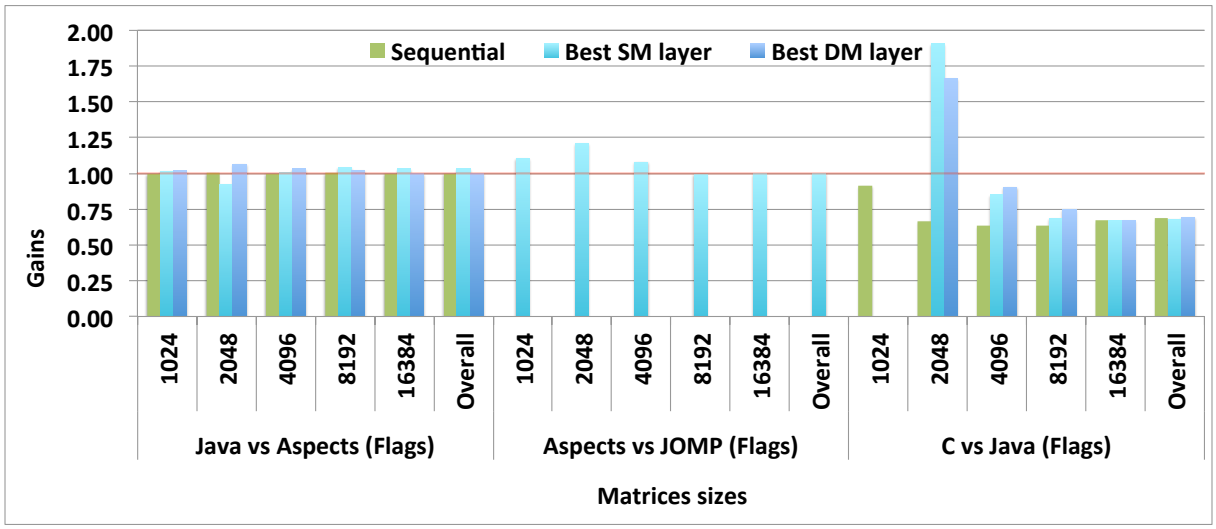


Figure 6.26: MM - Comparing different implementations of the sequential, SM, and DM versions.

[26] Namely, *-XX:+UnlockDiagnosticVMOptions*, *-XX:LoopUnrollLimit=500*, *-XX:-UseCompressedOops* and *-XX:ObjectAlignmentInBytes=256*.

In Figure 6.26 (and Table C.4), under the category Java *vs.* Aspects and the "Sequential" column, is possible to see that the design rules did not add overhead. In the SM and DM versions, we can also see that for the two smallest inputs Java was slower than C and that for the majority of the inputs the Java intrusive implementations were faster than the aspects, ranging from 0.92 to 1.06. The value of 0.92 for SM MM of matrices with size 2048 can be considered an outlier. Overall the SM Java intrusive was 1.03 faster than the aspects, while in the DM version there was virtually no difference (1.00). Compared with JOMP, the AOmpLib was faster for the smallest inputs (especially in the outlier), with gains up to 1.10, but overall with similar performance (1.00).

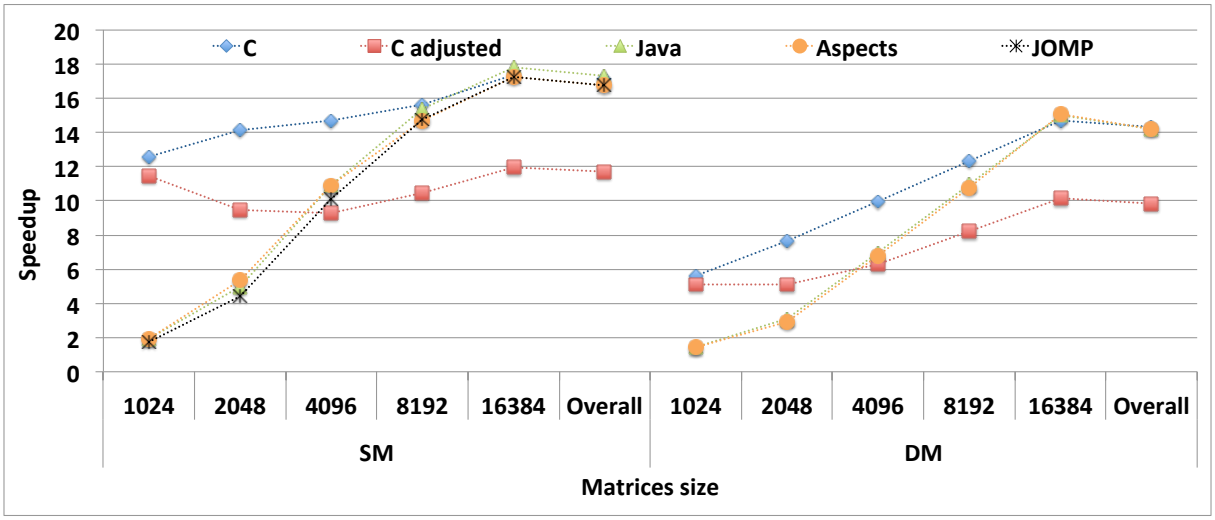


Figure 6.27: MM - Speedups of different implementations of the SM and DM versions.

Figure 6.27 presents the speedups of the different SM and DM versions (more detail in Tables C.8 and C.9). The blue curve of Figure 6.27 shows that for most inputs the C implementations scaled the most. However, the red curve ("C adjusted") shows that, except for the two smallest inputs, the Java-based versions were faster (with the help of the flags) than C. Since for the two largest inputs the Java intrusive implementations achieved speedups over 15.0×, we decided to test them in a multi-machine setup as well.

Figure 6.28 shows the speedups of the DM and hybrid versions in a cluster of 8 machines. Due to hardware constraints[27], the DM versions were tested with a maximum of 64 processes spread across 8 machines. Figures 6.27 and 6.28 (or more explicit Tables C.8, C.9, and C.11) show that the Java intrusive DM version for the two biggest inputs was able to scale, respectively, from 10.99 and 15.03× (in 1 machine) to 11.29 and 24.06× (in 8 machines). Moreover, for the same inputs, from the Java intrusive SM version to the correspondent hybrid, the speedups increased from 15.36 and 17.79× to 19.70 and 41.25×. Hence, executing our MM in multi-machines only pays off with the hybrids, and mainly for the last input.

[27]The Myrinet used is limited to 8 ports. Using Ethernet, in the MM case study, to communicate among more than 8 processes *per* machine introduced a prohibitive overhead.

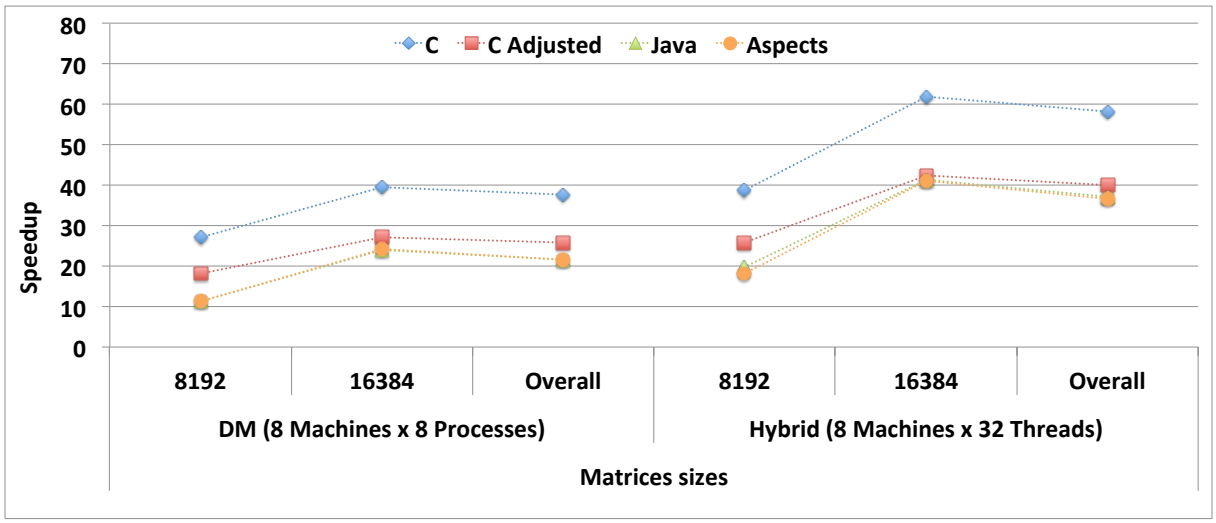


Figure 6.28: MM - Speedups of hybrids and DM versions with 8 machines.

Figure 6.29 shows that the hybrid versions were overall 1.54, 1.72, and 1.68 times faster than the pure DM in C, Java, and aspect implementations, respectively (details in Table C.13). The gains of the hybrids over pure MPI come mainly from a lower communication overhead (*i.e.*, 8 processes instead of the 64 used in the pure MPI), the ability to run in more cores (*i.e.*, 32 instead of 8) and, possibly, the dynamic balancing. With 8 threads *per* machine instead of 32, the hybrids are 1.05, 1.39, and 1.37 times faster than the pure MPI in C, Java, and aspect implementations, respectively. Therefore, the majority of the gains of the hybrid in C (*i.e.*, 90%) comes from the extra cores, whereas for the hybrids with Java and with aspects the extra cores did not contribute as much (*i.e.*, 46% for Java and 47% for the aspects). Nevertheless, these values are rough estimations, since for a more precise evaluation it would be required, among others, cache and load balancing values.

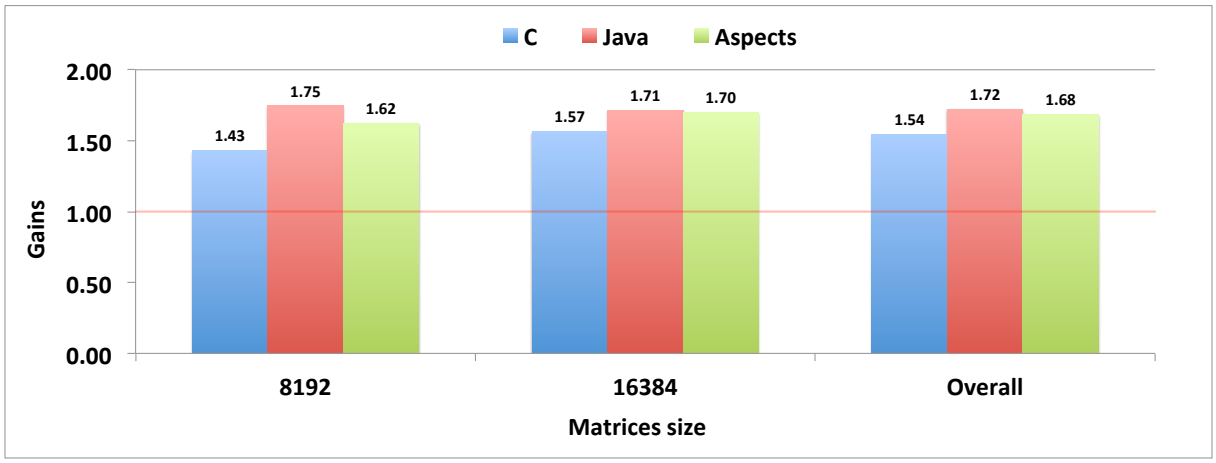


Figure 6.29: MM - Hybrids *vs.* DM versions with 8 machines.

Figure 6.30 shows the comparisons of different DM and hybrid versions in a cluster of 8 machines. Overall the Java intrusive implementations were 1.00 and 1.02 faster than the aspects in the DM and hybrid versions, respectively. The performance difference between the implementations with Java and with aspects was slightly more noticeable in the hybrids, following a similar trend to the results shown in the MD case study. By analyzing the *C vs. Java* comparison we can see that contrary to the single machine test results, Java was slower than C (even with the use of the JVM flags). The matrix layout of the C implementations allows performing fewer MPI process communications than the layout used in Java. Hence, in C the communication overhead was lower than in Java (results in Table C.6)).

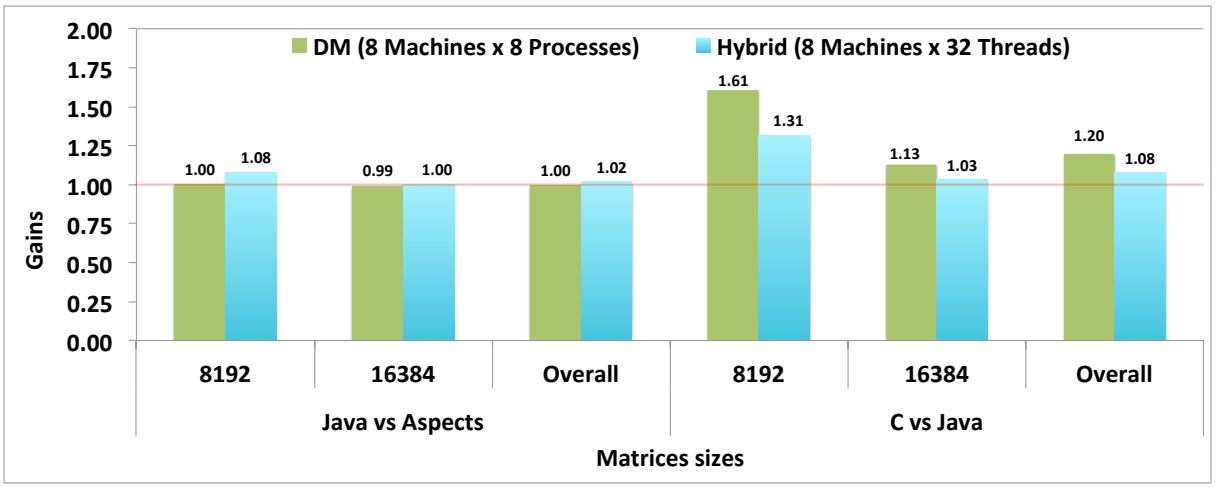


Figure 6.30: MM - Comparing parallel versions in DM/Hybrid with 8 machines.

6.4 Case Study : JGF Benchmarks

This section shows how our framework deals with a broader range of case studies and legacy code regarding performance and programmability compared to other approaches. The case studies presented in this section are the parallel benchmarks from section II and III of the JGF.

This section is structured differently from the *MD* and *MM* sections. We start with the introduction of all the algorithms, and then we present the improvements made to them. Afterward, we present the evaluation of the performance and programmability of our framework in comparison to other approaches.

6.4.1 Introduction of the case studies

We start by introducing the case studies with relevant information about their original JGF parallelizations. For some of the case studies, we found better strategies and optimizations. All those findings are properly indicated in the upcoming sections.

Crypt

This case study uses the International Data Encryption Algorithm (IDEA) [LMM91] to encrypt and decrypt data [SBO01]. The JGF implementation of this algorithm has three main data structures that hold the text to be encrypted, the encrypted and the decrypted data. In the most computational demanding routine, the data is first encrypted and then decrypted; both phases use an auxiliary sub-routine labeled "cipher_idea". The SM JGF implementation parallelizes the outer loop of the routine "cipher_idea" and the DM version splits the data to be encrypted/decrypted among processes. For the parallelization of this algorithm it is relevant to know that:

- the parallelized loop contains a loop-unrolling of degree 8;
- there is a dependency among iterations of the "cipher_idea" loop, that needs to be solved before parallelizing it. After that dependency is solved the algorithm is *embarrassingly parallel*;
- this algorithm is compute bound with an asymptotic complexity of $\Theta(\mathtt{N})$, with `N` being the size of the text to be encrypted.

Initially, this algorithm came with three inputs (3, 20 and 50 million elements) that ranged from 8.58 to 143.05 Megabytes, to these we added two additional ones (200 and 900 million) for a maximum of 2574.92 Megabytes.

Series

This algorithm computes the first $\mathtt{N}$ coefficients of a Fourier Series [Wei] for the function $\mathtt{f(x)} = (\mathtt{x}+1)^{\mathtt{x}}$ in the interval $[0, 2]$ [SBO01], which corresponds to the formula:

$$\mathtt{f(x)} = \frac{1}{2}\mathtt{a_0} + \sum_{\mathtt{n}=1}^{\mathtt{N}} \mathtt{a_n cos(n\pi x)} + \sum_{\mathtt{n}=1}^{\mathtt{N}} \mathtt{b_n sin(n\pi x)} \text{ [Wei]} \tag{6.1}$$

$$\mathtt{a_0} = \int_0^2 (\mathtt{x}+1)^{\mathtt{x}}\, \mathtt{dx} \tag{6.2}$$

$$\mathtt{a_n} = \int_0^2 (\mathtt{x}+1)^{\mathtt{x}}\, \mathtt{cos(n\pi x)}\, \mathtt{dx} \tag{6.3}$$

$$\mathtt{b_n} = \int_0^2 (\mathtt{x}+1)^{\mathtt{x}}\, \mathtt{sin(n\pi x)}\, \mathtt{dx} \tag{6.4}$$

The main kernel is composed by a section that first computes the term $\mathtt{a_0}$ and an outer loop that computes the terms of $\mathtt{a_n}$ and $\mathtt{b_n}$ from 1 to $\mathtt{N}$. These terms are computed using 1000 integrations steps (implemented with an inner loop). The outer loop iterations are independent of each other, and therefore the algorithm is *embarrassingly parallel*. The threads/processes of the SM/DM JGF implementations are responsible for computing a chunk of coefficients. The algorithm is compute bound with a complexity of $\Theta(\mathtt{N})$ and originally had three inputs of 10k, 100k, and 1 million elements, up to 15 Megabytes. We added two more inputs of 2 and 2.5 million elements, up to 38.52 Megabytes. Unfortunately, for this case study inputs that required more memory would not be convenient since it would, prohibitively, increase the execution time.

SOR

This case study solves a system of $\mathtt{X}$ linear equations using a variant of the Gauss-Seidel [GS00] iterative method, but with a relaxation factor (ω) of 1.25, which enables a faster convergence. Because ω is greater than 1 the method is called successive over-relaxation (SOR).

The system of linear equations is represented by a matrix $\mathtt{X} \times \mathtt{X}$. The main kernel of the SOR implementation of JGF has two loops that iterate over all positions of the matrix. That main kernel is executed 100 times (*i.e.*, convergence phase) and the content of the matrix is updated using the formula:

$$\mathtt{M_{ij}} = \frac{\omega}{4}(\mathtt{M_{i-1j}} + \mathtt{M_{i+1j}} + \mathtt{M_{ij-1}} + \mathtt{M_{ij+1}}) + (1-\omega)\mathtt{M_{ij}} \tag{6.5}$$

where $\mathtt{M}$ represents the matrix that holds the results. That formula includes the use of a stencil of four points, which implies the presence of dependencies between the loop iterations. Hence, this version is hard to be parallelized efficiently [SBO01]. Nevertheless, the JGF parallel implementations of the SOR use a "red-black" [Mit14] strategy that makes it easier to exploit

parallelism. In this version, the convergence phase is divided into two sub-phases named red and black, where odd and even rows numbers are updated, respectively. In the SM version, at the end of each phase, threads synchronize using a global barrier, whereas in the DM version processes will swap the rows of their borders using a blocking communication routine. Moreover, threads/processes have to be aware of their matrix borders, which implies extra conditional statements inside the main kernel. Since one of our goals is to use the same base code for all the parallelizations, we used a sequential base implementation based on the "red-black" approach. Although that decision did not negatively affect the performance, the code became less readable.

This algorithm is memory bound with an asymptotic complexity of $\Theta(\mathtt{N})$, with `N` being the matrix size. Initially, the inputs use were `1k` $\times$ `1k`, `1.5k` $\times$ `1.5k`, and `2k` $\times$ `2k` elements up to 30.52 Megabytes of memory. We added two more inputs of 10k and 15k elements that use up to 1716.61 Megabytes, to emphasize the memory bandwidth constraints of the algorithm.

LUFact

This case study solves a system of X linear equations using a modified version of a Gaussian elimination known as lower upper factorization (LUfact) proceeded by a triangular solve [SBO01]. The LUFact decomposes a given matrix `A` into a product of a lower (L) and upper (U) triangular matrices (`A` $=$ `LU`).

This case study is based on the original Linpack benchmark [DRM], which implements a LU factorization with partial pivoting. The system of linear equations is represented by a matrix `X`$\times$`X`. The most computational demanding routine of this case study is the row elimination phase, where for each column `k` of the matrix a vector-vector multiplication of the pivot column with the remaining columns is performed, starting with column `k` $+$ `1`. That phase is the main source of parallelism in LUFact since all vector-vector multiplications can be performed in parallel. However, the total loop iterations decrease as the factorization algorithm goes through the columns of the matrix since for each column `k` the row elimination performs `X` $-$ `k` $-$ `1` iterations.

This case study is memory bound and has an asymptotic complexity of $\Theta(\mathtt{N}^{\frac{3}{2}})$, with `N` being the matrix size. For the same reason as the SOR (to stress the bandwidth), we added two additional inputs of `8k` $\times$ `8k` and `16k` $\times$ `16k` for matrices with sizes up to 1972.82 Megabytes.

Sparse

This case study performs, a hundred times in a row, the multiplication of a vector by a sparse matrix stored in the format COO (*coordinate format*).

```
1 for (int r = 0; r < 100; r++)
2   for (int i = 0; i < N; i++)
3     y[row[i]] = y[row[i]] + x[col[i]] * val[i];
```

Figure 6.31: JGF - Sparse Kernel.

From all the case studies, Sparse has the simplest kernel (Figure 6.31). The vectors *row*, *col* and *val*, shown in Figure 6.31 together, represent the matrix saved in the COO format. The JGF SM implementation of this algorithm parallelizes the inner loop of Figure 6.31. To avoid race conditions during the update of vector y the vector *row* is sorted and the content of the vectors *col* and *val* are adjusted accordingly. Moreover, there is also additional logic to ensure that the same row is not assigned to different threads. In the DM implementation processes have a copy of the vector y and in-between iterations of the outer loop processes communicate to update their copies of the vector.

This algorithm is memory bound with irregular memory access patterns and an asymptotic complexity of $\Theta(\mathtt{N})$, with `N` being the number of matrix elements. We added two extra inputs of 5 and 7.5 million elements that use up to 137.33 Megabytes of memory.

MC

This case study uses Monte Carlo methods to simulate a financial environment [SBO01]. Among all case studies, MC has the highest number of classes. The sequential version of the MC has 15 classes, 145 methods, and 1145 lines of code[28].

The main kernel of the MC is composed by a loop with `N` iterations, with `N` being the input size. Every iteration of that loop executes a Monte Carlo simulation on a data that emulates a price stock and saves the results of the simulation in a vector of price stocks. The SM and DM parallelizations assign to threads/processes chunks of price stocks to be simulated. Although in the SM implementation the vector of results is shared among threads, there is no risk of race conditions because the vector data structure from Java internally uses synchronization during the critical operations (*e.g.*, add). Naturally, in the DM implementation, each process has its vector of results that at the end of the simulation is merged in a single vector.

The MC case study is compute bound and with an asymptotic complexity of $\Theta(\mathtt{N})$. Originally, this case study had only two case studies (10k and 60k elements), but we added three more (90k, 120k, and 150k elements) for a maximum of memory consumption of 2222 Megabytes.

RayTracer

This case study generates a scene with 64 spheres using a 3D Ray Tracer and saves the results in an array of size `N` [SBO01]. Similar to the MC, RayTracer is considerably larger than the other case studies, with a total of 13 classes, 52 methods and 548 lines of code[28]. The main kernel renders the scene and has a double loop that iterates through the x and y coordinates of the scene. In the JGF SM/DM implementations, threads/processes are responsible for rendering different parts of the scene. The SM parallelization needs to deal with the shared state and dependencies

[28]Statistics measured with CodePro AnalytiX .

inside the double loop. For instance, a shared variable named "checksum" accumulates the RGB elements of each pixel in the scene.

This algorithm is compute bound and performs several memory allocations and has an asymptotic complexity of $\Theta(N)$, with `N` being the number of pixels in the scene. This case study currently uses five different inputs (three added by us) ranging from `150x150` to `2500x2500` elements for a maximum of memory consumption of 606 Megabytes.

6.4.2 Performance and programmability evaluation

The size of the inputs used (*i.e.*, total of elements and memory occupied by them) in the JGF case studies are presented in Tables E.1 and E.2 from the Appendix E. The execution times of the sequential, SM, and DM versions of all the implementations (C, Java, JOMP, and aspects) are shown in Tables E.3, E.5, and E.6, respectively.

6.4.2.1 Improvements

The improvements made to the original JGF implementations came either from better strategies found with our framework or straightforward improvements. These improvements are not only related to increasing performance but also with improving the code readability.

Concerning the sequential versions, the performance of the RayTracer and SOR was significantly improved. Figure 6.32 (and Table E.4) presents the gains from these improvements.

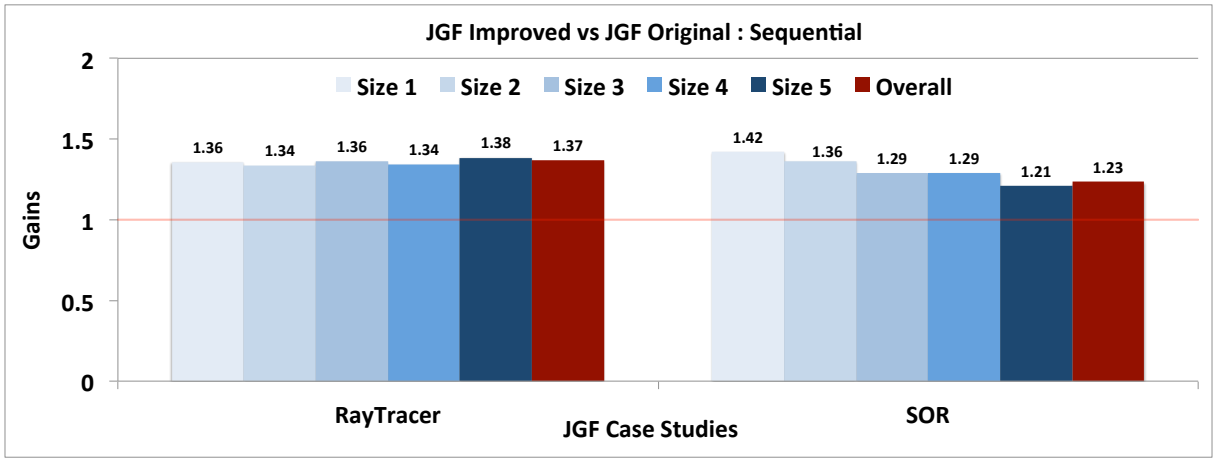


Figure 6.32: JGF - The gains of improvements in the sequential versions.

The JOMP version of the RayTracer does not use the original base code from JGF. The JOMP implementation replaced, in the base code, fields that were susceptible to race conditions by local variables passed as parameters of the methods that used these fields. Hence, resulting in several modifications to the (original) base code. We speculate that these fields were replaced due to the inability of JOMP to make them thread private. Nevertheless, the JOMP version with a single thread was overall 1.37× faster than the original JGF sequential version. Hence, we

used the JOMP sequential implementation as the new (sequential) base code. Compared with the original version the new implementation produces fewer instructions, occupies less memory, and allocates fewer objects (details in Table F.1 of Appendix F).

We improved the SOR sequential version (the one based on the red-black algorithm) by reducing the number of conditional tests. The snippet of code of Figure 6.33 shows the structure of the SOR kernel, which consists of applying a stencil over a matrix. To apply the stencil (lines 9 to 11) the SOR kernel needs to differentiate among the first, last, and middle rows.

```
1   ...
2   public void sor_simulation(...)
3   {
4     ...
5     for(int p = 0; p < JACOBI_NUM_ITER; p++)
6     {
7       for(int i =  1 + (p % 2); i < MAX_ROWS; i += 2)
8       {
9         if(/* First row */ ) {...}
10        else if( /** Last row **/) {...}
11        else {...} // Middle rows
12      }
13    }
14  }
```

Figure 6.33: JGF - SOR Kernel.

Our optimization consisted in moving the first and last row tests to outside the loop. Consequently, we had to adapt the loop range to exclude the first and last rows. We further applied the same optimization over the stencil heuristics themselves (not included in Figure 6.33). Worth noting that although our improvement reduced the number of instructions, it also increased the absolute number of misses to the cache L2 and L3 (details in Tables F.2 and F.3). As shown in Figure 6.32 this optimization provided an overall gain of 1.23. Naturally, we implemented the parallel versions of the RayTracer and SOR upon their improved sequential versions.

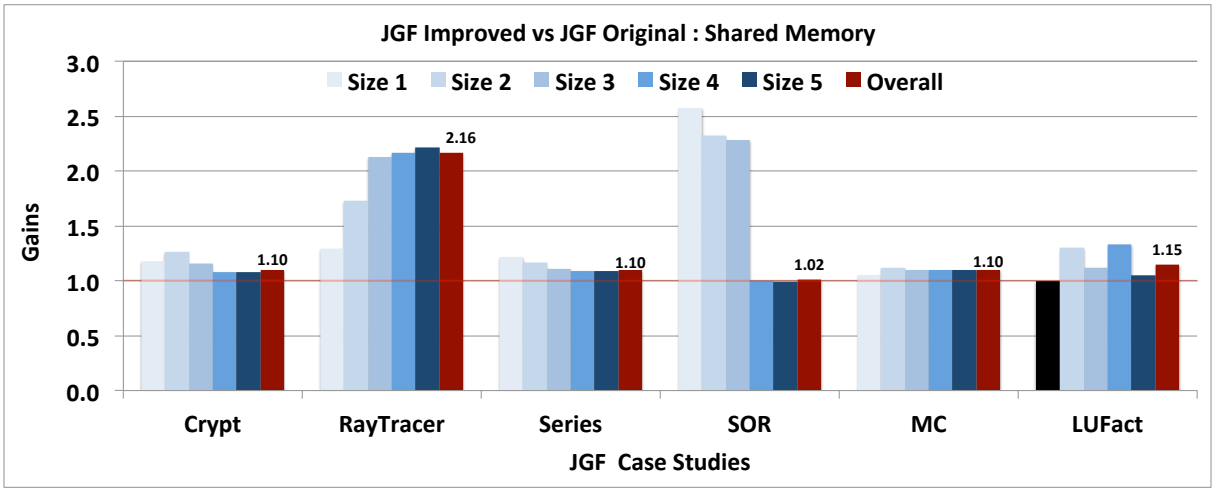


Figure 6.34: JGF - Gains of the improved SM versions.

After the improvement of the sequential versions of the case studies, we tackle their SM versions. Figure 6.34 presents the gains obtained for the improved SM versions (detailed in Table E.7). The improvements made over Crypt and SOR were the only ones that did not come directly from the tests made with our framework. In Figure 6.34 for the smallest input of the LUFact, there is a black bar to illustrate that we did not consider that input because for that size the parallel version of the LUFact had a speedup under one.

The SM version of Crypt originally had two consecutive parallel *for* constructors, one *per* encrypting and decrypting phases. We improved that version by creating a single parallel region that wraps the encrypting and decrypting parallel *fors*. This improvement reduced the number of threads/runnables created and the number of calls to synchronization mechanisms (*e.g.*, *join*). Furthermore, in the original version, the heuristic that distributes the iterations of loops *per* threads was calculated twice (encrypting and decrypting) even though both parallel *fors* had the same range, which means that with static distributions threads will work with the same range in both *fors*. With this in mind, we tested different *for* distributions with our AOmpLib and concluded that a static distribution with default chunk was the fastest. As shown in Figure 6.34 those improvements provided an overall gain of 1.10 (0.284 seconds).

The improvements in the SM versions of the RayTracer, Series, MC, and LUFact came from tests made with the AOmpLib. The use of a dynamic *for* distribution (with a chunk of one) alone improved the RayTracer, Series, and MC case studies. Nonetheless, we also added the following improvements to the RayTracer: - sharing some objects among threads instead of making them private; - reducing the synchronization to update a shared variable (named *checksum*)[29]; - the use of the changes made by JOMP in the (sequential) base code, among others. These improvements provided an overall gain of 2.16 (31.918 seconds), from these the AOmpLib contributed directly with 1.19 (9.571 seconds). The use of the dynamic distribution in the Series and MC delivered in both an overall gain of 1.10 that corresponds to a reduction of 4.151 and 0.566 seconds, respectively.

The LUFact had two different SM parallelizations, the original from the JGF and the one provided by JOMP. In the JGF version a parallel region wraps around the entire method that performs the factorization and inside of it there is a *for* distribution in the row elimination routine. In this version, the master thread executes most of the operations inside the parallel region sequentially. Moreover, inside that parallel region, six synchronization barriers are required to ensure that the threads wait while the master is working. In the JOMP version, there is only a parallel *for* during the row elimination. At first glance, the main advantage of the JGF approach is that there is the creation of only one parallel region, whereas in the JOMP approach there are multiple parallel regions (one for each time the row elimination method is invoked). However,

[29] In the improved version threads have private copies of the *checksum* variable and at the end of the parallel region, all the *checksum* variables, from the different threads, are reduced.

with thread pools, threads can be created once and reused afterward, instead of being created every time a parallel region is reached.

After testing several approaches with the AOmpLib, including the two aforementioned, results showed that overall the JOMP approach is 1.15× faster than the original JGF approach. Moreover, the JOMP approach is also less complex since it only has one parallel *for* constructor, whereas the JGF approach has a parallel region, a *for*, 6 *barriers* and 3 *masters* constructors. It is worth mentioning that the improved version benefited more with the Java *NUMA* flag[30] than the original version (details in Table E.11). Without the *NUMA* flag, the LUFact improved version would only have significant gains in the three smallest inputs, those that fit in cache.

Regarding the SOR, the gains of its SM and DM versions, shown in Figures 6.34 and 6.35, are the result of the (previously mentioned) reduction of conditional tests. In the SOR SM and DM versions, the gains are higher in the three smallest input sizes (varying from 1.61 to 2.57) and practically nonexistent in the remaining two. That results from the fact that: - first, the two largest inputs do not fit in the cache; - second, with the improved version the absolute and percentage L3 cache misses decreased with the three smallest inputs and increased with the remaining two (results in Tables F.4 and F.5); - and, finally, the parallel versions have a different bottleneck. The parallel versions of the SOR (and also the LUFact) are limited by the memory bandwidth, which is most noticeable for inputs that do not fit in cache. Nonetheless, the SM and DM improvements still reach an overall gain of 1.02 and 1.03, respectively.

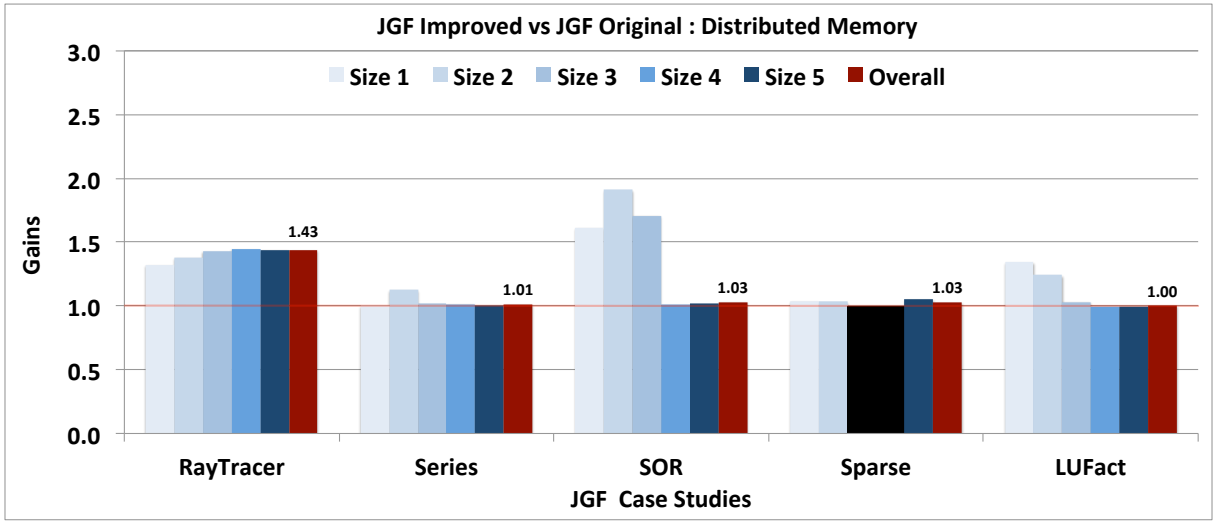


Figure 6.35: JGF - Gains of the improved DM versions.

Figure 6.35 and Table E.8 present the gains of the improved DM versions. The overall gain of 1.43 (12.146 seconds) from the RayTracer DM improved version, as with the SOR, came from the optimizations performed in the sequential version.

[30]The flag *-XX:+UseNUMA* is a Java JVM flag used to make the Parallel Scavenger garbage collector NUMA-aware [Fla]. For simplicity, we refer to this flag as *NUMA* flag.

The improvements in the Sparse DM version came from the phase when we compared the intrusive approach to our layer of aspects. During that phase, the tests showed that the layer of aspects was overall faster than the matching Java intrusive implementation (*i.e.*, the original Sparse DM version). Contrary to the AOdmLib, the JGF version used a global array as a temporary array during the reduction[31] of the matrix across processes. We replaced that global array by a local one, which provided an overall gain of 1.03 (0.672 seconds).

```
1   public void calculatingFourierFirstNparis (...){
2     ...
3     // Calculate the fourier series. Begin by calculating A[0].
4     TestArray[0][0] = TrapezoidIntegrate(...);
5
6     for(int i = 1; i < TestArray.length; i++)
7     {
8       // Calculate A[i] terms.
9       TestArray[0][i] = TrapezoidIntegrate(..., omega * i, 1);
10      // Calculate the B[i] terms.
11      TestArray[1][i] = TrapezoidIntegrate(..., omega * i, 2);
12    }
13  }
```

Figure 6.36: JGF - Series Kernel.

The improvements made in the DM versions of Series and LUFact focused more on reducing the complexity of their parallelizations than increasing performance. Figure 6.36 presents a code snippet[32] extracted from the sequential version of Series. The Series SM version distributes among threads the loop iterations shown in line 6 of Figure 6.36. However, in the DM version, each process allocates a smaller sub-matrix of the global matrix (labeled "TestArray" in Figure 6.36) where the elements of the trapezoid integrations are stored. In the end, the slaves send their chunks of columns to the master process. Figure 6.36 shows that the index (i) of the loop is being used as the series number (lines 9 and 11). Consequently, in the original JGF DM version processes have to convert the indices of their local matrix into the corresponding indices of the global matrix. Although we tested (using our layers of aspects) different DM versions, we did not find one that considerably outperformed the performance of the original. Nevertheless, one of them was slightly faster, and simpler, than the original version. This version uses a similar approach to the SM version, where every process has a global view over the matrix and parallelizes the loop from line 6. With this approach, there is no need to convert between local and global indices since the loop is aligned with the matrix access pattern. Moreover, to collect the results from all processes, we used a reduction routine with the master process as the root instead of using the more complex communication heuristic[33] present in the original version. With this new approach, we reduced the number of DM-related statements in the intrusive

[31] An *All_reduction* communication routine is called 200 times.

[32] The method labeled *calculatingFourierFirstNparis* in Figure 6.36 is named *DO* in the JGF.

[33] Sending a matrix by columns with *MPI send* requires more effort than a *MPI reduction*, especially in Java where there is no pointer arithmetic.

version from 44 to only 16, a reduction of more than half of the statements. The new approach provided an overall gain of 1.01 (0.517 seconds). The gains of the majority of the inputs vary between 1.00 and 1.02, except for the second input. For that input size, the gain was 1.13 because for that input the reduction communication routine outperformed the one used in the original version.

Similar to Series, the original LUFact DM version divides the matrix among processes, however, in the latter, the division is performed by rows instead of columns. Moreover, the original LUFact DM version uses auxiliary structures to map the processes ids to the indices of the rows that each process should work. The use of these structures originated in part from the need to translate between global and local matrix indices. Similar to the improvement made in Series, in the LUFact improved DM version processes have the entire matrix instead of only a subset of rows. The combination of this new approach with some additional arithmetic enables the removal of the auxiliary structures along with several lines of code. Furthermore, because the matrix was divided by rows, we applied a further optimization to minimize memory usage. With our additional optimization processes allocate the entire matrix, but the rows that will not be touch by them are set to *null*. Hence, with this additional optimization, we combined the advantage of having the entire matrix (*i.e.*, no need to convert between global and local indices) with the advantage of the original version (*i.e.*, avoiding the duplication of the matrix in all processes).

The improved approach reduced almost to half the number of DM related-statements (from 86 to 46). Besides simplifying the code, this new approach also uses less memory not only because of the removal of the auxiliary structures but also because the master process does not need to keep a sub-matrix along with the global matrix. Overall the performance of the new approach is virtually the same as the original; nevertheless, for the three smallest inputs we achieved gains ranging from 1.03 to 1.34 (up to 0.027 seconds). The improvement is only noticeable in the three smallest inputs because for the two biggest the LUFact takes considerable more time (15 to 193 times more) and the reduction in the execution time provided by the improvements does not scale with the input size.

The previously mentioned feature of setting rows to *null* based on some matrix distribution is also provided by our AOmpLib transparently to the user.

6.4.2.2 Framework evaluation

Design Rules

Before showing the parallel implementations, we start by presenting in Table 6.7 the design rules and metrics required to implement them. These design rules include all the methods and modifications needed to test the different layers of parallelism – including setters and gathers added to access (from the layers) private fields of the base code. Some of these design rules could have been removed after the best layer was found, but we kept them because it provides potential useful join points. Worth mentioning that not all parallel constructors forced the introduction of new design rules, some of them reused methods that either initially existed in the base code or resulted from previous design rules (column labeled "# of join points reused" in Table 6.7).

Unsurprisingly, most of the design rules for the SM layers were *for* methods, which in the case of RayTracer, Series, and MC were reused in their DM layers as well. The DM layers of all case studies used the *object creation* design rule, sometimes more than once, which was expectable since we created layers to test parallelism based on explicitly decomposing the domain data.

Table 6.7: JGF - Summary of the design rules applied.

Case Studies	SM Design Rules	DM Design Rules
Crypt	For;	OCDR (3); Method;
RayTracer	For;	OCDR; Method;
Series	For;	OCDR; Method (2);
SOR	For;	OCDR; Method (5);
Sparse	Method; A.I. (2);	OCDR (4); Method (3);
MC	For;	OCDR; Method (2);
LUFact	For; Method (3);	OCDR; Method (2);
Total	For (6); Method (4); A.I. (2);	OCDR (12); Method(16);

Case Studies	# of DR	# of join points reused	Collateral Design Rules	# of new Methods	# of new Statements
Crypt	6	6	1	4	13
RayTracer	4	5	1	3	8
Series	5	3	1	4	11
SOR	6	7	1	5	15
Sparse	10	12	-	8	17
MC	4	4	-	4	9
LUFact	8	5	-	8	16
Total	43	42	4	36	89
Average	6.1	6	0.6	5.1	12.7

DR - **D**esign **R**ules;
For - *for* method design rule;
A.I - **I**njection of an **a**rgument into a method;
OCDR - **O**bject **C**reation **D**esign **R**ule;
Method - creation of a method to provide a join point;
(X) - 'X' is the number of times such a rule was applied. When omitted X=1.

In some cases, besides the typical design rules, additional refactoring was needed ("collateral design rules" in Table 6.7). In the entire JGF benchmark only four collateral design rules were added, three concerning parallel loop correctness (Crypt, RayTracer, and SOR) and one related with domain decomposition (Series). Regarding the latter, we already described a similar issue in the MM case study. In Series, a field was being used to hold the number of columns of a matrix, which could lead to problems since that number can vary at runtime due to AOdmLib data management features. To preempt any future issues we replaced that variable by explicit accesses to the data structure size, which resulted in the refactoring of a single statement[34]. Regarding the loop correctness, the refactoring applied in the Crypt, RayTracer, and SOR case studies had also been done in the JOMP versions. The refactoring was done because the body of those loops could not be executed in parallel and not because of limitations in JOMP or our framework – without these changes, the loops would be invalid in OpenMP as well.

We argue that most of the design rules applied made the code more readable since they introduced well-named methods with well-defined behavior. Nevertheless, we also acknowledge that four of them might look slightly unnatural at first, namely two design rules regarding access to data indices and two methods wrapping around conditional conditions. These design rules were necessary due to the restrictions on the AspectJ join point model regarding the interception of accesses to array positions and conditional conditions. Therefore, to be able to intercept and manipulate these types of join points we created methods out of them. We needed to know the accessed positions in the data structures of the RayTracer and Series to allow conversions between local and global data views indices. The wrapping of the conditionals was used in the SOR because the heuristics to detect the first and last iterations can vary with the parallelization. The creation of methods out of those conditions enables the layers of aspects to intercept them and adapt the conditions accordingly. Nevertheless, we could have avoided the wrapping of the conditions by coding them directly in the base code. However, that would go against our philosophy of having a unique base code regardless of the parallelization in place.

Sparse and LUFact were the case studies that required the highest number of design rules. In the Sparse, we applied four times the *object creation* design rule, and in the LUFact its original SM implementation required the introduction of six *barriers* and three *masters* constructors. In the latter, most of the newly created methods could have been inlined since the best SM layer found only needed the application of a *for* method design rule.

Figure 6.37 shows that for the majority of the inputs the overhead of the design rules is almost nonexistent. With exception of LUFact, the gains of the code without design rules over the one with design rules ranged from 0.99 to 1.01 with 1.00 (no overhead) being the overall result. As expected, adding methods or exposing parameters did not add a significant overhead.

[34] Although, in the end, the best DM layer of Series did not rely on any data management features we still kept that refactoring in the source code.

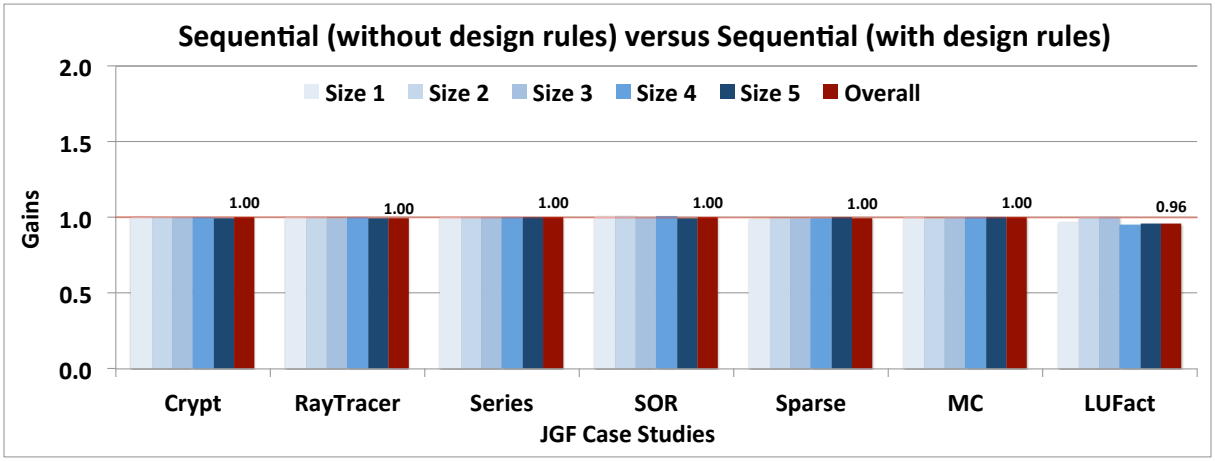


Figure 6.37: JGF - The gains of the sequential code with and without design rules.

Overall, in the LUFact the design rules reduced 2.998 seconds from its original execution. With the introduction of the design rules, a variable in the routine *daxpy* is kept in a register, whereas previously it was allocated in the stack. This unexpected event reduced the number of load instructions and, consequently, the execution time. Figure F.1 and Table F.6 show the assembly and profiling results of the LUFact design rules, respectively.

Apart from the LUFact the other unexpected behavior caused by the design rules happened during the layer testing phase for the Sparse case study. During that phase, the introduction of the *for* method design rule increased the overall execution time of the sequential version in 5.948 seconds (an increase of 1.19×). After analyzing the assembly of the sequential code with the *for* method design rule, we verified that the statement *i+=step* was generating 3 instructions (2 *mov* and 1 *add*), whereas the version without that design rule (the one using the statement *i++* instead) generated only 1 (*inc*). The assembly of both versions can be seen in Figure F.2. We removed the *step* from the *for* method since the best SM/DM layers did not need it.

SM Layers

Tables 6.8 and 6.9 show for the best SM Layers the parallel constructors used and the number of statements needed to implement them, respectively. As shown in Table 6.8 aside from Sparse all case studies used a *for* constructor. The original JGF Sparse parallelization uses, instead, a customized loop distribution, which was reused in the best layer as well. That distribution only modifies the *begin* and *end* of the loop based on the thread ID. Hence, the use of two constructors to inject different argument values into a method (labeled as "A.C." in Table 6.8). The other constructors worth mentioning are: - in the RayTracer, the creation of a private object *per* thread along with the reduction of a field from it and making four other fields shared (shown in Figure F.3); - in the SOR, the replacement of the AOmpLib default barrier by a user-defined one through the use of Java' method overriding mechanism (Figure F.4).

Table 6.8: JGF - Constructors of best SM Layers.

Case Studies	Parallel constructors used in the Best Layer
Crypt	P.R.; For Static;
RayTracer	P.For Dynamic; Private Object;
Series	P.For Dynamic;
SOR	P.R.; For Static with user-defined barrier;
Sparse	P.R.; A.C. (2);
MC	P.For Dynamic;
LUFact	P.For Static;

P.R. - **P**arallel **R**egion;
P.For - **P**arallel **F**or;
A.C. - **C**hanging the value of an **A**rgument from a method;
(X) - 'X' is the number of times such a constructor was used. When omitted X=1.

Table 6.9 shows, under the AOmpLib category, the number of statements needed to apply the design rules (column labeled "DR") and to implement the layer of aspects using a pointcut-based approach (column labeled "Pointcuts") and alternatively using an annotation-based approach (column labeled "Anno."). For the JOMP/OpenMP implementations, we included the number of statements needed for both the annotations and for the additional intrusive code (column labeled "Non-annotation."). Finally, the column labeled as "Ratio" shows the ratio of the number of statements used in the Java intrusive implementation to the number of statements needed with a pointcut-based approach (including the design rules).

Table 6.9: JGF - The number of statements needed to implement the best SM Layers.

Case Studies	AOmpLib			JOMP/OpenMP		Intrusive	Ratio
	Pointcuts	Anno.	DR	Anno.	Non-annotation		
Crypt	7	3	5	2	1	28	2.3
RayTracer	13	6	4	1	19	20	1.2
Series	7	2	3	1	1	13	1.3
SOR	17	-	5	1	21	38	1.7
Sparse	50	-	3	1	44	51	1.0
MC	7	2	3	1	0	14	1.4
LUFact	7	2	3	1	0	23	2.3
Total	108	15	26	8	86	187	1.4
Average	15.4	3	3.7	1.1	12.3	26.7	1.6

DR - **D**esign **R**ules;
Anno. - Annotations;
Ratio - Intrusive / (Pointcuts + DR).

Unsurprisingly, as shown in Table 6.9, the Java intrusive implementation required more statements than the other approaches. The Java intrusive implementations needed, in total, approximately 1.4× the number of statements used by the AOmpLib with a pointcut-based approach. That number increases to approximately 2.9× when compared with AOmpLib annotations[35].

[35]Since we did not use annotations in the SOR and Sparse case studies, we did not count those when comparing the number of statements used in annotations with those used in the intrusive. Hence, the intrusive and annotations used 98 and 33 statements, respectively.

In the previous values (*i.e.*, pointcuts and annotations) we also included the number of statements needed for the design rules. Sparse was the case study with the lowest ratio between the number of needed statements with the Java intrusive implementation and with AOmpLib. Regardless of the used implementation (*i.e.*, AOmpLib, JOMP/OpenMP, or intrusive), the Sparse required several statements because all of these implementations use the original customized loop distribution (this routine alone has 35 statements). The fact that with AOmpLib we had to implement from scratch the customized loop distribution explains why the AOmpLib used almost the same number of statements as used with the Java intrusive implementation.

SOR, Crypt, and LUFact were the case studies with intrusive implementations that used more than 1.5× the number of statements used by AOmpLib implementations. These case studies, along with Sparse, are the only ones that use a static loop distribution. In our Java intrusive implementations, the coding of the static loop distributions required more statements than the dynamic loop distributions.

Although we reduced the number of statements of all JGF SM parallelizations – for instance, instead of using classes for the runnables we used lambdas in the scope of the method applying the parallelism – that number is still significant, mostly because of loop heuristics (static by blocks). Nonetheless, that number can be reduced with the use of utility-type classes that centralize the commonly used heuristics (*e.g.*, loop distributions). However, such a detailed study is out of the scope of this work.

As excepted the implementations using annotations have the fewest number of statements, especially those from OpenMP/JOMP. Unfortunately, sometimes annotations are not enough and consequently have to be complemented with additional code. For instance, the JOMP/OpenMP implementations of the RayTracer, SOR, and Sparse required, besides the introduction of the annotations, other modifications to the base code. We refactored the RayTracer because of JOMP's restrictions and the Sparse case study because of its specific loop distribution. The SOR was refactored because of its tailor-made *barrier* and because with our improvement to its sequential code it is no longer feasible to use the *for* annotation. Consequently, in the SOR JOMP/OpenMP implementations, we coded the loop distribution instead of using annotations.

If we consider only the AOmpLib layers that use annotations and compare them with the annotations from JOMP, including the refactoring needed in both, AOmpLib required approximately 1.2× more statements. Unlike the annotations from JOMP, the annotations from our framework have to be encapsulated into an aspect, which counts as an additional statement.

Before analyzing the performance results of the best SM layers, it is worth to mention that the flag *NUMA* from Java was used in the SOR, LUFact, and Crypt case studies since it improved their performance. Moreover, the Series case study was executed with Java 1.9, since it is considerably faster with that version than with Java 1.8 (more than 5×), more details will be provided later in this chapter.

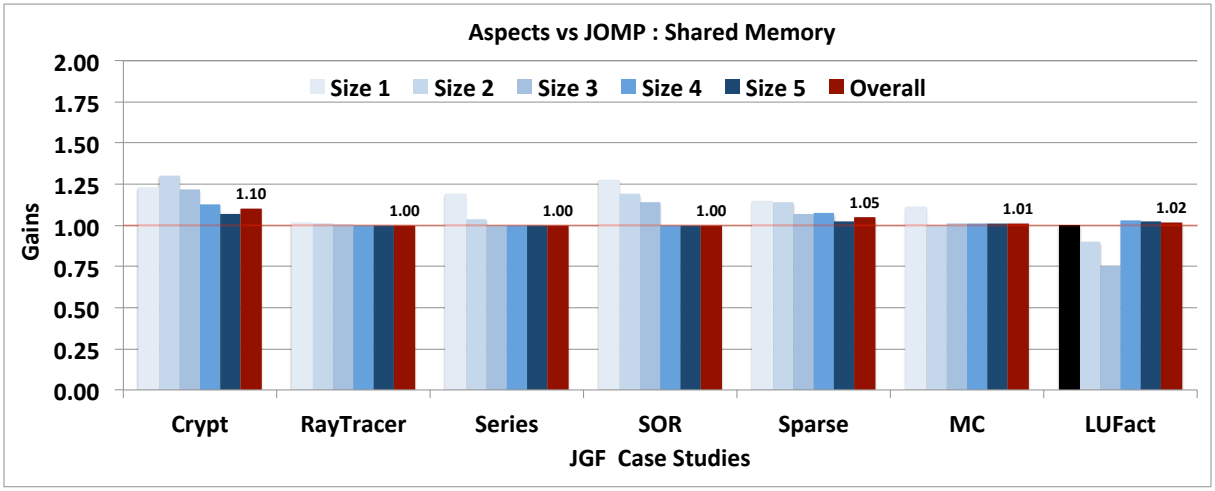


Figure 6.38: JGF - The gains of the Aspects *vs.* JOMP.

Figure 6.38 (and Table E.10) shows the gains of AOmpLib over JOMP. For the inputs with execution times under 0.12 seconds the gains of AOmpLib over JOMP are typically higher but decrease with the increase of the input size/execution time. That indicates that, usually, the fixed overhead of the parallel constructors from JOMP is higher than the ones from the AOmpLib. Therefore, unsurprisingly, the AOmpLib considerably outperforms JOMP in the two case studies with the lowest SM execution time (Crypt and Sparse with execution times under 2.5 seconds for any input). Compared with the AOmpLib, JOMP introduced more instructions in the Crypt and Sparse case studies and produced in the Crypt case study more accesses and misses in the L2 and L3 caches (details in Tables F.9, F.10, F.11, and F.12).

The LUFact is, performance-wise, very sensitive to code changes in its kernel, as we could verify in the design rules overhead (Figure 6.37). In this case study, for the second and third smallest inputs, JOMP was 0.017 and 0.116 seconds faster than AOmpLib, corresponding to gains of 0.90 and 0.75, respectively. These differences are related to the fact that JOMP uses a different synchronization barrier from the one used by AOmpLib. JOMP uses their barrier implementation inspired in the *tournament barrier* algorithm proposed by Hengsen *et al.* [HFM88], whereas AOmpLib uses a *cyclic barrier* introduced in Java 1.5. The barrier implementation used in JOMP is the same as the one used in the original JGF benchmark, which according to [Pet11][36] outperforms the *cyclic barrier* in the LUFact case study. We tested the JOMP implementation using the *cyclic barrier* instead of the *tournament barrier* and, indeed, it became slower. Compared with that JOMP implementation AOmpLib is overall 1.05× faster and is now 1.15× and 1.06× faster in the second and third smallest inputs, respectively.

Although in the LUFact, JOMP outperformed the AOmpLib in the second and third inputs it was not able to do the same in the two largest inputs. For the two largest input sizes two factors influenced the differences between JOMP and AOmpLib, namely the flag *NUMA* and

[36] At that time the tests where performed with JVMs older or equal to 1.6.

the ratio of the input sizes to the number of barrier calls. In all SM implementations of the LUFact, the flag *NUMA* significantly reduced the execution time of the last two inputs (details in Table E.11). With that flag, in the two largest input sizes, the JOMP implementation became 1.60 and 1.50× faster, whereas the AOmpLib became 1.66 and 1.47× faster. Without the flag *NUMA*, in the last two inputs, the AOmpLib would have been 0.99 and 1.04× faster than JOMP, respectively. Finally, compared with the second and third inputs, the last two inputs had a lower ratio of the input size to the number of barrier calls. Hence, benefiting the AOmpLib.

The performance difference between *cyclic barrier* and *tournament barrier* was only noticeable in the LUFact because, apart from the SOR, it is the only case study that performs several barrier calls. Although SOR also performs several barrier calls, it uses a different, and simpler, version of a barrier, which was also used in the AOmpLib and JOMP implementations. For all the other case studies we have replaced the original *tournament barrier* for a *cyclic barrier*. We did that refactoring because it makes it easier to compare the results from the JGF and AOmpLib implementations and because the *cyclic barrier* is an implementation [Pet11] that is provided by Java standard API. Nevertheless, as experienced in the SOR, changing the default barrier of AOmpLib is just a matter of overriding a method in the concrete layer of aspects.

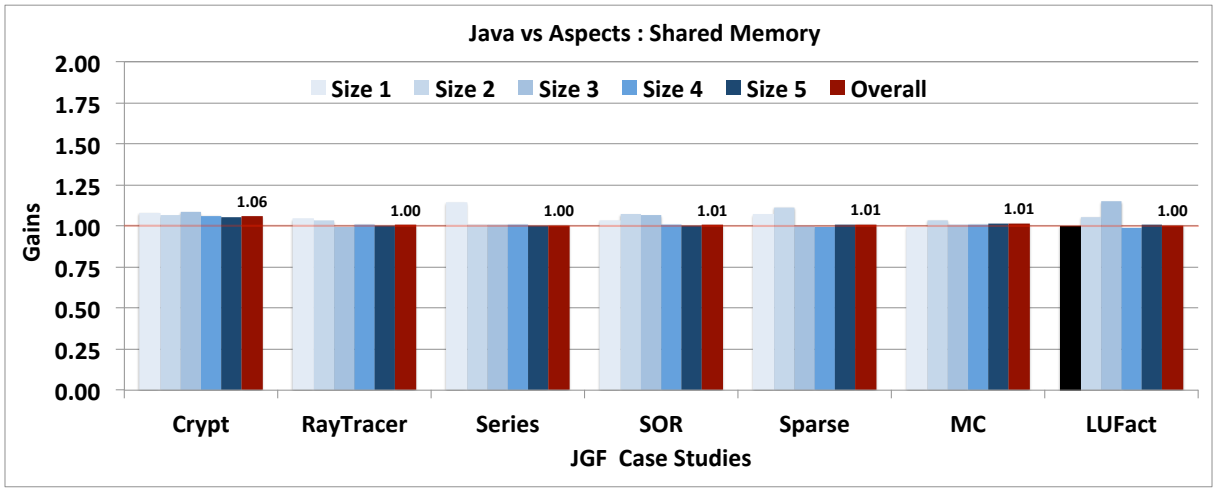


Figure 6.39: JGF - The gains of the Java intrusive *vs.* Aspects in SM.

Figure 6.39 shows the gains of the intrusive Java implementations over the AOmpLib. Analyzing that figure we can extrapolate similar conclusions to the ones made in the AOmpLib *versus* JOMP comparison. Apart from Crypt, the gains of the intrusive versions over the AOmpLib are higher for the smaller inputs, typically under 0.12 seconds, but these gains decrease with the increase of the execution time. The AOmpLib parallel constructors, naturally, have a higher overhead than those added intrusively. Moreover, the overhead of the AOmpLib constructors is typically fixed, hence not increasing with the input size of the case study. Therefore, percentage-wise this overhead is more noticeable in the smaller inputs. Nevertheless, in the majority of the case studies, overall AOmpLib had an equivalent performance to the Java intrusive.

The Crypt was the case study where the AOmpLib performed the worst compared with the intrusive implementation. From all the case studies, Crypt has the lowest execution time, thus emphasizing, the most, the AOmpLib overhead. Moreover, our profiling results show that the AOmpLib added more instructions, cache accesses and misses (detail in Tables F.13 and F.14).

The SOR and LUFact case studies are memory bound and took advantage of the flag *NUMA*. Overall, the flag *NUMA* reduced 12.324 and 9.026 seconds from the SOR and LUFact SM intrusive Java implementations, corresponding to gains[37] of 1.68 and 1.55, respectively. As shown in Figure 6.39, the SOR and LUFact Java intrusive implementations outperform the AOmpLib in the sizes that fit in the cache (the three smallest sizes). In these case studies and input sizes, the AOmpLib added more cache accesses/misses, which among other factors explains the performance differences between the AOmpLib and the intrusive versions. The profiling results are presented in Tables F.7 and F.8.

DM Layers

Table 6.10: JGF - Constructors of best DM Layers.

Case Studies	Parallel constructors used in the best DM layer
Crypt	Master (2); Scatter (4); Gather; GlobalView(3);
RayTracer	Master; Scatter; Gather; LocalIndex; For Static; Reduce;
Series	Master(2); For Static; Reduce;
SOR	Master; Scatter (2); Gather; GlobalView; SendRecv(2); UDA(3);
Sparse	Master(3); Scatter (6); GlobalView(4); AllReduce; DataDuplication(3);
MC	Master (2); Scatter; Gather; For Static;
LUFact	Master(5); Scatter (2); Gather; Ignore; Bcast; UDA;

(X) - 'X' is the number of times such a constructor was used. When omitted X = 1;
UDA - **U**ser-**D**efined **A**dvices.

Tables 6.10 and 6.11 show the parallel constructors and the number of statements needed to implement the best DM Layers, respectively. As shown in Table 6.10, all case studies, apart from Series, used the scatter and gather constructors provided by AOdmLib data partitioning module. These constructors were further tuned, varying from just ensuring that the data was correctly aligned (*e.g.*, Crypt with an alignment of 8 elements) to complete scatters and gathers implemented from scratch (*e.g.*, SOR). The other constructors worth mentioning are: - the distribution of data by chunks and the use of constructors to converts from global to local indexes (Figure F.5) in the RayTracer case study; - the scatter and gather of vectors (Figure F.6) in the MC case study.

[37] Gains of the versions with the flag *NUMA versus* the versions without flag *NUMA*.

The best SOR DM layer is one of the two that used user-defined *advices* (three in total). Two of these *advices* intercept the methods (resulted from the design rules) that wrap around the logic to detect the first and last rows based on the borders of the SOR matrix. Since in the best DM layer the matrix is scattered across processes the logic to detect the first and last rows differs from the one in the base code. These user-defined *advices* ensure that the logic from the base code to detect the first and last rows is adapted to the DM parallelization. The other user-defined *advice* intercepts the method resulting from the *for* design rule and adapts its range based on the current phase (*i.e.*, red or black).

```
1   public int dgefa(...){
2     ...
3     for(/** each column **/)
4     {
5       if(/** process to work **/){
6         ...
7         // update pivot
8         if(/** if this column is not yet triangularized **/ ){
9           // some work
10          // compute multipliers
11          row_elimination(...);
12        }
13        else {...}
14      }
15      // Broadcast data
16      row_elimination(...)
17    }
18    ...
19 }
```

Figure 6.40: JGF - LUFact main kernel.

Besides SOR, LUFact also used user-defined *advices* (only one) to replicate part of the logic of the original JGF DM parallelization (illustrated in Figure 6.40). In Figure 6.40, the red lines of code represent the DM parallelization, the scratched gray line the code that was removed from the sequential implementation and the remaining lines the code common to the sequential and DM implementations. Regarding the implementation with the layer of aspects, the difficult part of replicating the original JGF parallelization is the move, in the base code, of the *row elimination* method call from one point to the other (from the line 11 to the line 16).

During the test phase of the SM and DM layers, a method design rule was applied over the lines 8 to 13 of Figure 6.40 (method named *triangularized*). The best DM layer (part of it shown in Figure 6.41) intercepts the call to the *row elimination* method (line 11 of Figure 6.40) and ensures that no process will execute that method. After the call to the *triangularized* method, the user-defined *advice* (line 8 of Figure 6.41) injects the remaining missing logic (lines 15 and 16 of Figure 6.40), which includes the call to the *row elimination* method (line 14 of Figure 6.41).

```
1   ... static aspect filter extends Dm_Filter {
2         pointcut ignore() : call (... void row_elimination (...))
3                             && withincode(... triangularized(...))
4     }
5
6   ... static aspect injectL extends Dm_Injection ...{
7
8     ... around() : call(... triangularized(...)) && target (source) && ...{
9
10        if(/** process to work **/){
11          ... result = proceed(...);
12        }
13        // Broadcast data
14        source.row_elimination(...);
15        return result;
16      }
17  }
```

Figure 6.41: JGF- The best DM layer of the LUFact case study.

The SM and DM layers of the LUFact were able to correctly compose with the same base code, notwithstanding, we argue that transformations as the one introduced by the user-defined *advice* can threat the composition of layers to form the hybrids. The *row elimination* method resulted from the application of the *for* method design rule during the testing phase of the SM layers. The best SM layer intercepts that method and parallelizes its loop. The problem with the user-defined *advice* from the DM layer is that it moves the call of a method, that is intercepted by the SM layer, from the base code to the DM layer. Depending on how the programmer restricted the pointcut in the SM layer that intercepts the *row elimination* method call, that pointcut might not intercept any method call after a hybrid parallelization. For instance, if the programmer restricted that method call based on the target object (*call(row_elimination(...) && target(Linpack)*), in a hybrid parallelization, the SM layer would not be able to intercept the *row elimination* method call performed from the DM layer.

Table 6.11: JGF - The number of statements needed to implement the best DM Layers.

Case Studies	AOdmLib			Intrusive	Ratio
	Pointcuts	Anno.	DR		
Crypt	26	-	8	42	1.2
RayTracer	20	15	4	28	1.2
Series	12	5	4	17	1.1
SOR	61	-	10	70	1.0
Sparse	31	-	14	55	1.2
MC	11	9	6	55	3.2
LUFact	38	-	10	46	1.0
Total	199	29	56	313	1.2
Average	28.4	9.7	8.0	44.7	1.4

DR - **D**esign **R**ules;
Anno. - Annotations;
Ratio. - Intrusive / (Pointcuts + DR).

Table 6.11 shows the number of statements necessary to implement the best DM parallelizations with the AOdmLib and intrusively. By comparing Tables 6.11 and 6.9 we can see that the best DM parallelizations of most case studies required more statements than the best SM parallelizations, regardless if implemented with AOdmLib or intrusively. The best DM parallelizations of all case studies with our libraries required 1.9× more statements than their best SM parallelizations, while intrusively it required 1.7× more. This difference between the number of statements comes mainly from the fact that most of the best DM parallelizations use explicit domain data decomposition, which either intrusively or with the layers of aspects requires more statements than the best SM parallelizations (that rely mainly on loop parallelism).

For the SM parallelizations of all case studies, the intrusive implementations required 1.4× more statements than the layers of aspects, while for the DM parallelizations that value decreased to 1.2. This decrease is related to the fact that for most case studies their best DM layers were more customized (*i.e.*, tuned) and required more design rules than their best SM layers. The design rules applied during the implementation of the DM layers used 2.2× more statements than those of the SM layers, even though the DM layer reused some of the design rules introduced during the SM testing phase. Hence, this emphasizes that the DM versions required more code restructuring than the SM versions. Both consequences (*i.e.*, more tuning and design rules) resulted from the fact that the majority of the best DM layers used the AOdmLib data partitioning module. Usually, the use of this module requires the customization of the handling of the target data (*e.g.*, scatter by lines), which corresponds to the creation of methods in the concrete layer of aspects that override the ones from the abstract layers. Hence, this further increases the number of used statements. Furthermore, applying the *local/global data view* feature over a data structure will, typically, require the introduction of a few methods in the base code (*e.g.*, *object creation* design rule), which also increases the number of used statements.

SOR and LUFact were the only case studies where the number of statements used in their DM intrusive and AOdmLib implementations was almost the same. The best AOdmLib implementation of those case studies used user-defined *advices*, which naturally require more statements than using the default parallel constructors. Concerning the SOR almost half of the statements of its DM intrusive parallelization were used to implement the scatter and gather routines. These routines, by default, are not provided by the AOdmLib. Therefore, the AOdmLib default scatter and gather constructors were overridden, in the DM layer, by the same scatter and gather used in the intrusive implementation, which reduced the difference between the number of statements used in the intrusive implementation and the AOdmLib. In the LUFact the intrusive and AOdmLib implementations needed almost the same number of statements mainly because of the user-defined *advice* that replicates the DM logic presented in Figure 6.41.

The Sparse and MC were the only case studies in which the ratio between the number of statements used in their intrusive implementations and used in their layer of aspects increased from the SM to the DM parallelizations. Regarding the Sparse case study, its best SM layer was mainly composed by a customized loop distribution, while its best DM layer used parallel constructors that our framework provides by default. Since the DM layer did not use highly customized parallel constructors, that layer was able to save more statements (compared with the respective intrusive implementation) than the SM layer was. Concerning the MC case study, its best DM parallelization uses routines that are not provided by the OpenMPI implementation that we used in the intrusive implementations (*i.e.*, scatter/gather of vectors). The intrusive implementation of these routines resulted in several statements, which are provided by AOmpLib with significantly fewer statements.

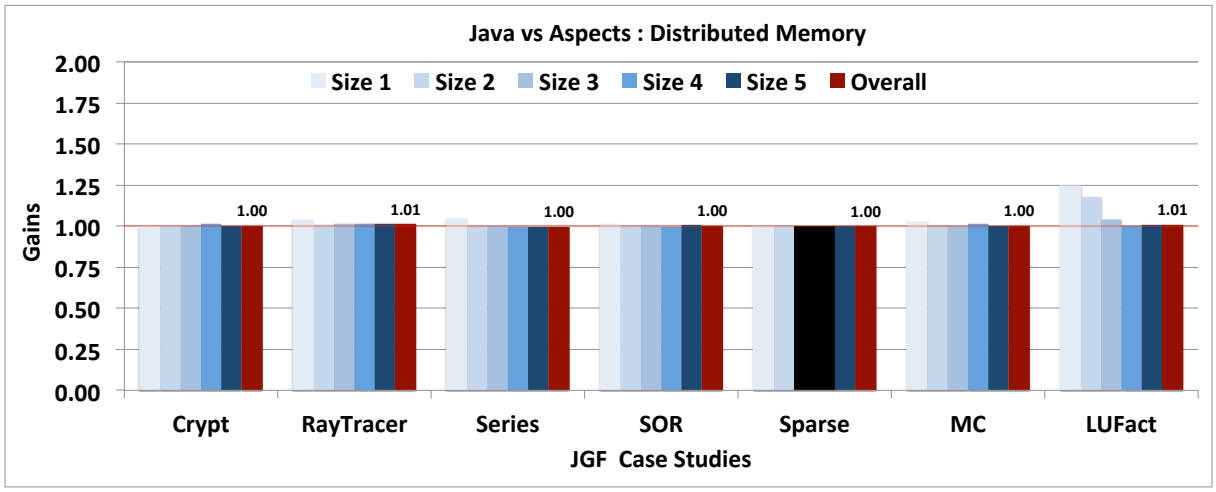


Figure 6.42: JGF - The gains of the Java intrusive *vs.* Aspects in DM.

Figure 6.42 presents the gains of the Java intrusive DM versions *versus* the AOdmLib. By comparing Figures 6.39 and 6.42, we can see that for most of the case studies the gains of the intrusive implementations over the libraries of aspects were higher in the SM parallelizations. Hence, this suggests that for most of the case studies the AOmpLib had a higher additional overhead than the AOdmLib. We expected that the gains of the Java intrusive implementations over the libraries of aspects would be higher in SM than in DM since the implementations of the parallel constructors with our framework or intrusively can vary more in the SM than in DM. However, not all case studies meet our initial expectation. Nevertheless, besides the parallel constructors of the libraries by themselves, it is also necessary to consider additional code changes that each may have imposed in the base code. For example, the base code of the DM implementation of the RayTracer has an extra design rule that does not exist in the SM implementation. That extra design rule[38] was added to allow the DM library to convert between local/global indices and may influence the execution time of the case study.

[38]That design rule added an extra division operation that will be executed as many times as the input size.

The LUFact was the only case study where the intrusive implementation was visibly faster than the AOdmLib in multiple input sizes. For the three smallest inputs of the LUFact, the intrusive implementation obtained gains ranging from 1.04 to 1.25, corresponding to a difference between execution times up to 0.020 seconds. From all the best aspect layers of all case studies, it was expected that the layer from the LUFact had the highest overhead (compared with the intrusive implementations) since it is the most complex. Moreover, that layer changes the position of the call to the row elimination method and performs that change in a code region that is executed multiple times.

Hybrid Layers

Figure 6.43 shows the speedups of the best SM and DM Java intrusive implementations for all the case studies. There are four case studies with SM versions that achieved speedups above 15×, namely Crypt, RayTracer, Series, and MC. From these case studies, only the RayTracer and the Series have also speedups above 15× in their DM versions. Hence, we created hybrid versions out of these two case studies and tested them with the inputs that also scaled above 15× in the SM and DM implementations. Tables E.13 and E.14 present in more detail the speedups of the SM and DM versions in C, Java (original and improved), Aspects, and JOMP.

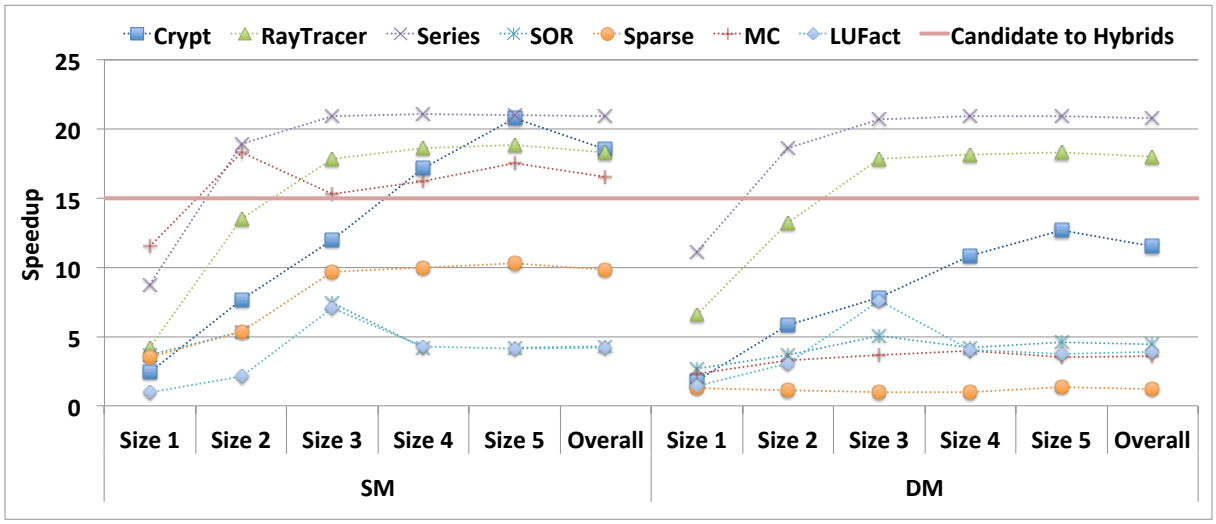


Figure 6.43: JGF - The speedups of the best Java intrusive parallel versions.

Similar to what we experienced in Sections 6.2 and 6.3 with the MD and MM case studies, implementing the hybrid versions of the RayTracer and Series with our libraries was just a matter of adding the SM and DM layers to the same build. As mentioned before, join points that are intercepted simultaneously by the SM and DM layers can threaten the correct composition of these layers. In the RayTracer and Series, the methods resulting from the *for* design rule are the only join points (on each case study) intercepted by pointcuts of both layers. In the RayTracer the *for* method is intercepted by six parallel constructors, three of each layer. From the SM

layer, it is intercepted by a parallel region, a dynamic *for*, and a creator of private objects, while from the DM layer it is intercepted by a static *for*, a gather, and a reduction. According to the precedence rules of our framework, each process will first create a team of threads (*i.e.*, parallel region) and then the private objects. Afterward, the iterations of the *for* method are (statically) divided among processes and then further (dynamically) divided among threads. After the call of the *for* method, and without any specific order, the appropriate data structures are gathered and reduced among processes. The compositional process of the Series hybrid aspect version was similar to the one described in the RayTracer. However, the Series did not have the creation of private objects from the SM layer and the gather from the DM layer. For both cases studies, the framework correctly and transparently dealt with the order on which the previously described code transformations were applied to the base code.

Similar to what has occurred in the MD and MM case studies, implementing the hybrid versions of the RayTracer and Series intrusively, mainly, required merging the SM and DM parallelizations with adaptations to the dynamic SM *for* to reflect the DM static *for*.

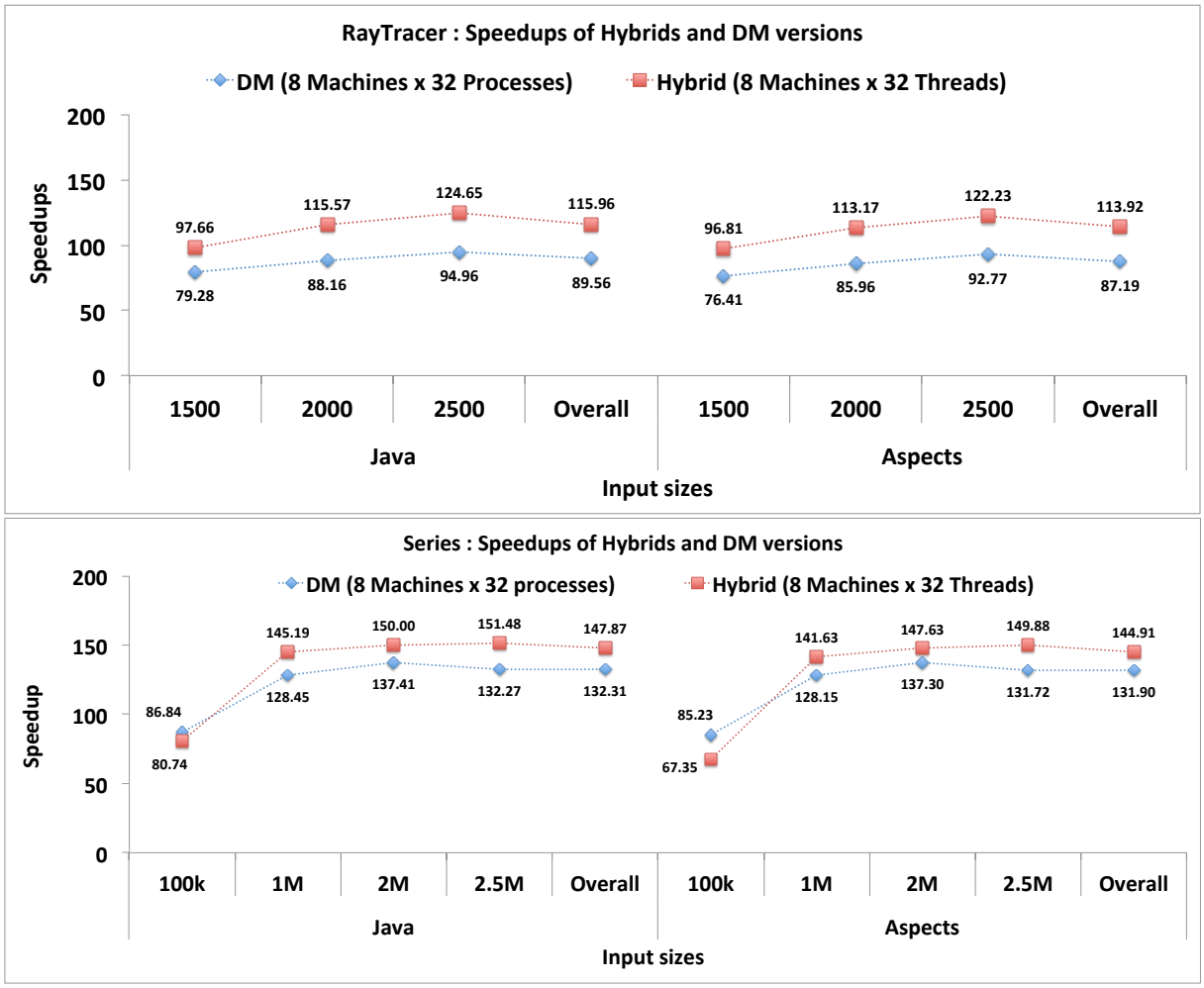


Figure 6.44: JGF - RayTracer and Series: The speedups of the DM/Hybrid with 8 machines.

Figure 6.44 shows the speedups of the DM and Hybrid versions of the RayTracer and Series case studies in a cluster with 8 machines. More details about these speedups and the correspondent execution times are shown in Tables E.15, E.16, E.17, and E.18. With exception of the 100k input size of Series, all hybrids outperformed the versions using only processes. For the majority of the input sizes, in both the intrusive and versions with aspects, the speedups of the DM and hybrids increased with the size of the inputs. Overall, in the RayTracer and Series, the intrusive hybrids achieved speedups of 115.96 and 147.87×, whereas the intrusive DM versions achieved speedups of 89.56 and 132.31×, respectively. The speedups of the versions with aspects followed the same trend as the intrusive versions, however with slightly lower speedups.

Figure 6.44 shows that Series, regardless of the version, achieved higher speedups than the RayTracer, following the same trend as the SM/DM versions of these case studies in a single machine. The difference between the scalability of the two case studies can be explained by the fact that: - the kernel from Series is more CPU intensive than the RayTracer; - the SM/DM parallelizations from Series are less complex and with lower overhead than those from RayTracer.

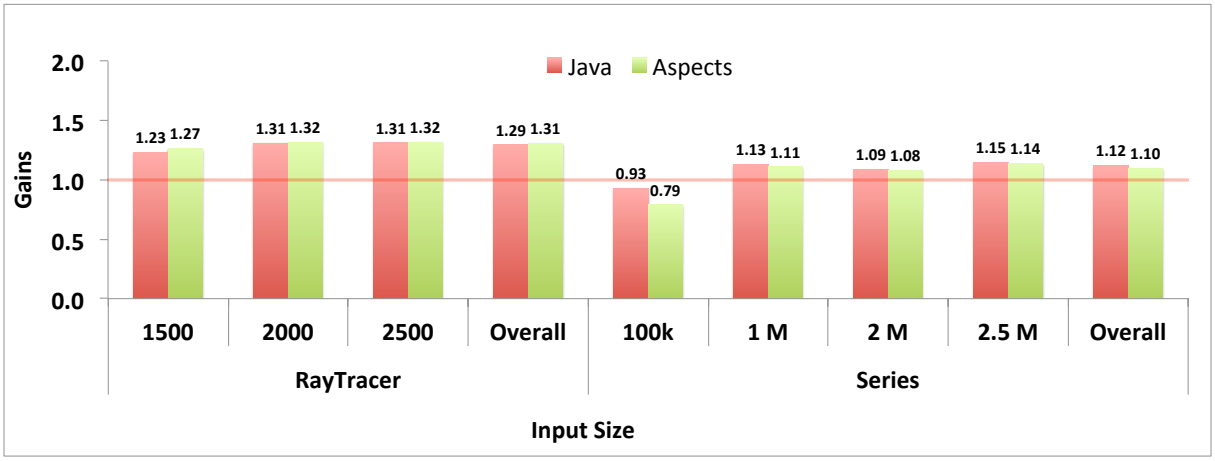


Figure 6.45: JGF - RayTracer and Series : The gains of the Hybrids *vs.* DM with 8 machines.

Figure 6.45 (and Tables E.19 and E.20) shows, for the RayTracer and Series, the gains of the hybrids over the DM in 8 machines. Overall the gains of the hybrids over the DM versions were higher in the RayTracer with its intrusive and aspect versions achieving gains of 1.29 and 1.31, respectively. The gains of the hybrids from the RayTracer were higher than those from Series because the hybrids from the RayTracer extracted more out of the hyper-threading than their corresponding DM versions. For 128 processes, 16 *per* machine, the overall gains of the (intrusive) hybrids over the DM versions in the RayTracer and Series were 1.05 and 1.00. While during hyper-threading (32 threads) the RayTracer intrusive hybrids achieved an extra 0.14 in gains, whereas the Series achieved 0.10. The hybrids from RayTracer and Series required significantly less memory than their DM versions. In the tests with 8 machines, for the largest input, the DM and hybrid versions of the RayTracer used around 9 and 2 Gigabytes of memory *per* machine, whereas the DM and hybrids of Series only used 3 and 0.1 Gigabytes, respectively.

The hybrids from the RayTracer and Series scaled more than the DM versions because the hybrids: - use Myrinet, whereas the DM versions only use Myrinet up to 8 processes *per* machine and from that point on Ethernet; - need less memory than the DM versions; - benefit from the SM dynamic distribution; - exploit more efficiently the hyper-threading than the DM versions.

The input with 100k elements of Series was the only one where the hybrids did not outperform the DM versions. That behavior also occurred in the single machine tests, however with a different input (10k). With a careful look at the speedups of the SM/DM versions of Series in a single machine (presented in Figure 6.43), we can see that the DM version only outperforms the SM version[39] in the smallest input (10k). Although 100k is an input 10× larger than 10k, in the tests with the cluster there are 8× more machines. Hence, an input of 100k in 8 machines corresponds to only 12.5k elements *per* machine, which explains why, for that input, the tests in multi-machines followed the same trend as those of 10k in a single machine.

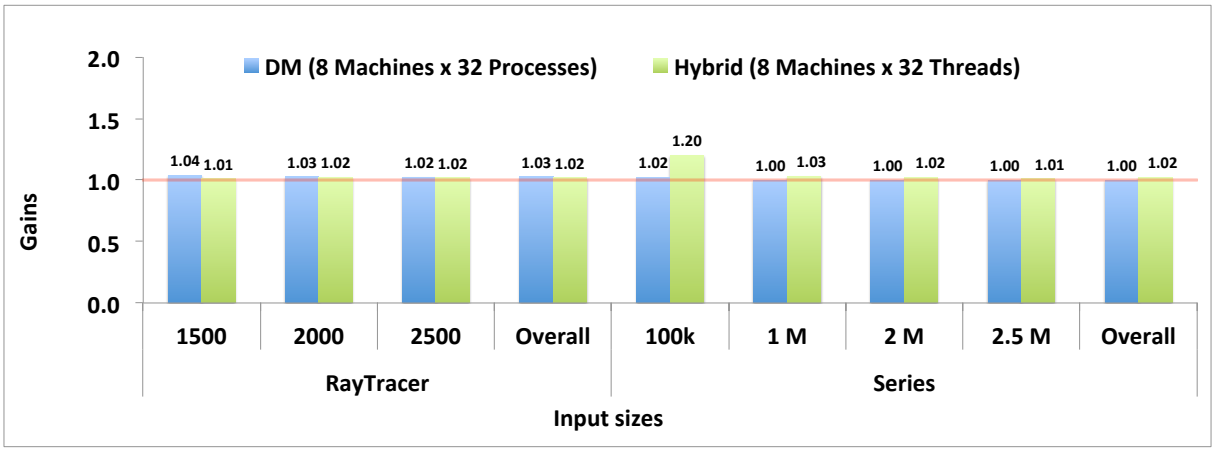


Figure 6.46: JGF - RayTracer : The gains of the Java *vs.* Aspects in DM/Hybrid with 8 machines.

Figure 6.46 shows the comparison of the DM/hybrid versions implemented intrusively *versus* implemented with our framework of aspects. These results follow the same trends as the ones from the SM/DM tests performed in a single machine (shown in Figures 6.39 and 6.42). In the single machine tests, the intrusive SM and DM versions of the RayTracer were 1.00 and 1.01 faster than our libraries of aspects, whereas the intrusive SM and DM versions of the Series had the same performance. In the cluster, the gains of the intrusive versions over the libraries of aspects increased compared with those in the single machine. However, the difference between the execution times of the intrusive and aspect versions did not vary significantly. If we aggregate the execution times of the four largest inputs, the Series SM and hybrid aspect versions are 0.177 and 0.126 seconds slower than the corresponding intrusive versions, respectively. However, because the execution time of the hybrids is significantly lower than those from the SM version (approximately 7×), the overhead of our libraries is more noticeable.

[39] A speedup of 11.11× from the DM version and 8.79× from the SM version.

In the RayTracer and Series, the intrusive implementations were 1.00 to 1.04× faster than our libraries of aspects (except for the input size of 100k of Series). Although the gain of 1.20× (corresponding to a time difference of 0.034 seconds) of the intrusive hybrid over the hybrid with our framework in the Series 100k input size may come as a surprise, in fact, it follows the same trend as the one from the single machine tests. As explained before, the input of 100k corresponds in the tests with multi-machines to 12.5k elements *per* machine. If we recall the results of the comparison between Java intrusive *versus* AOmpLib for the size of 10k the gain was also noticeable (1.15× corresponding to a time difference of 0.016 seconds). Furthermore, the difference between the execution times of the intrusive hybrid and the hybrid with aspects was practically the same for all inputs, varying from 0.029 and 0.034 seconds. However, because the execution time of the input with 100k elements is much lower than the remaining inputs (from 6 to almost 16× lower), the overhead of our libraries is more exacerbated by that input.

6.4.2.3 C vs Java

The RayTracer and the MC were the only case studies that we did not implement in C because these case studies use Java intrinsic features (*e.g.*, collections and inheritance) that would be hard to replicate precisely in C. Furthermore, as previously stated, comparing Java with C is not one of the main goals of this thesis. Nevertheless, we present in Figures 6.47, 6.48, and 6.49 the comparison of Java *versus* C in the sequential, SM, and DM environments, respectively. Although the C implementations replicate as much as possible the Java implementations, there are peculiarities that can affect their execution time, namely: - in the C implementations of the SOR, Series, and LUFact, the matrices were continuously allocated in memory, whereas in the Java implementations these matrices were allocated as arrays of pointers; - the Java implementations of the Crypt, SOR, and LUFact used the flag *NUMA*, whereas their C implementations exploited the *first-touch* policy on the initialization of their main data structures [NAAL01].

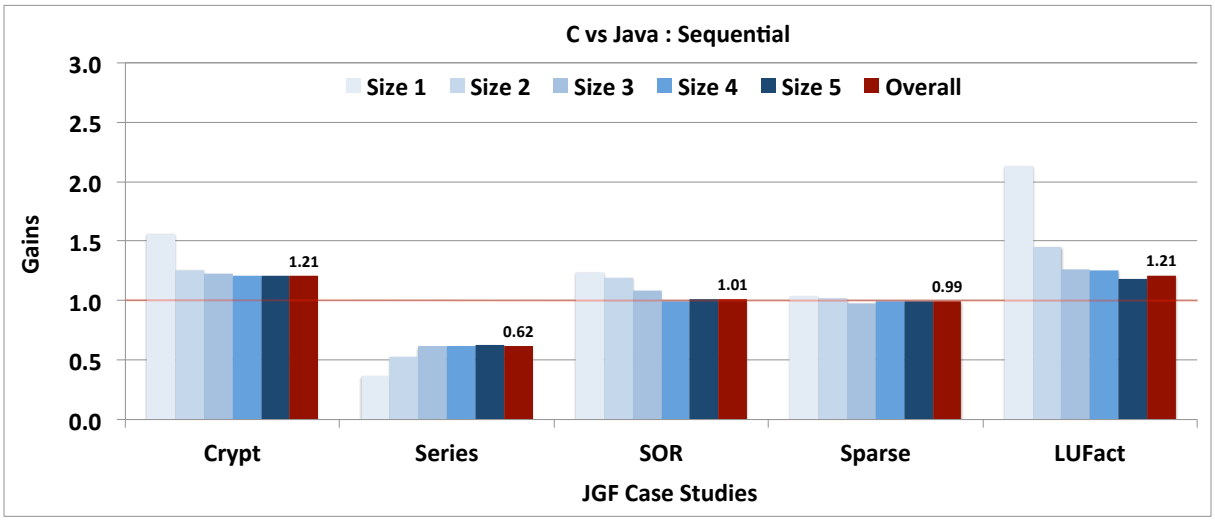


Figure 6.47: JGF - The gains of the C *vs.* Java in sequential versions.

The comparison between C and Java, for the Crypt, SOR, and LUFact, shows trends similar to the ones of previous comparisons (*e.g.*, Intrusive *versus* Aspects); the gains are high for the smaller inputs and decrease with the increase of the input. Moreover, as encountered in other comparisons, when there are significant changes in the SOR or Crypt implementations, these changes are noticeable in all inputs of Crypt and (mostly) in the three smallest inputs of SOR.

C outperformed Java the most in the Crypt and LUFact with an overall difference between the execution times of C and Java of 9.332 and 12.646 seconds, respectively. In both case studies, the Java implementation produced significantly more instructions than C, from 1.19 up to a maximum of 2.25× (*i.e.*, the first input of LUFact) more instructions. Tables F.15, F.16, F.17, and F.18 present in more detail the profiling results of Crypt and LUFact.

In the Sparse case study both Java and C implementations have overall approximately the same execution time, 30.476 and 30.637 seconds, respectively. Although in the Sparse for the two smallest inputs C was faster than Java, for the input sizes that do not fit in the cache the difference in execution time was practically none; three factors contributed to that behavior. First, the implementations are sequential therefore the code of both C and Java are very similar. Second, the sequential version of Sparse is very simple (presented in Figure 6.31). Third, the algorithm is memory-bound with an irregular memory data access pattern.

Series was the only case study with Java implementations (*i.e.*, sequential, SM and DM) that outperformed the C implementations. Series, as mentioned before, was executed with Java 9, which made its sequential code overall 5.77× faster than executing it with Java 8. The version with Java 9, unlike Java 8, exploited vectorization. Series is compute bound and relies heavily on functions from the *Math* library, namely, *pow*, *sin*, and *cos*, which were greatly improved with Java 9. Unfortunately, to the best of our knowledge, at the time of the writing of this document no official technical paper was released explaining those improvements in detail. Nevertheless, a talk about the improvements made in the Java *Math* library can be found in [VV17]. When executing the Series sequential code with Java 8, C was overall 3.58× faster than Java. However, when executing with Java 9 C was 1.61× slower than Java. As with Java 8, and unlike Java 9, none of the Series C implementations exploited vectorization. Therefore, just as the sequential version, the parallel versions of Series implemented with C were slower than the ones implemented/executed with Java 9.

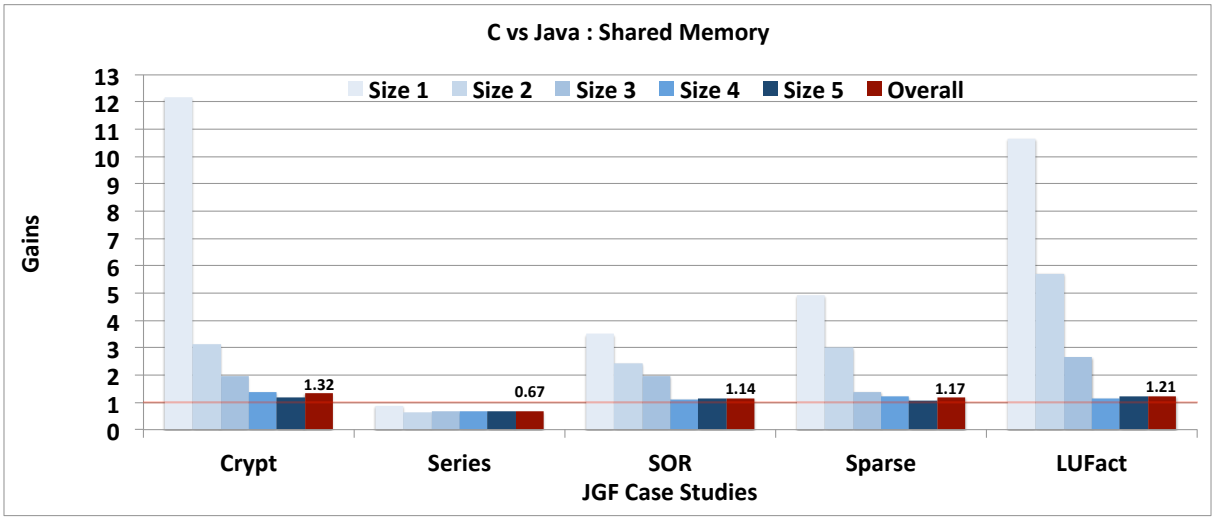


Figure 6.48: JGF - The gains of the C *vs.* Java in SM versions.

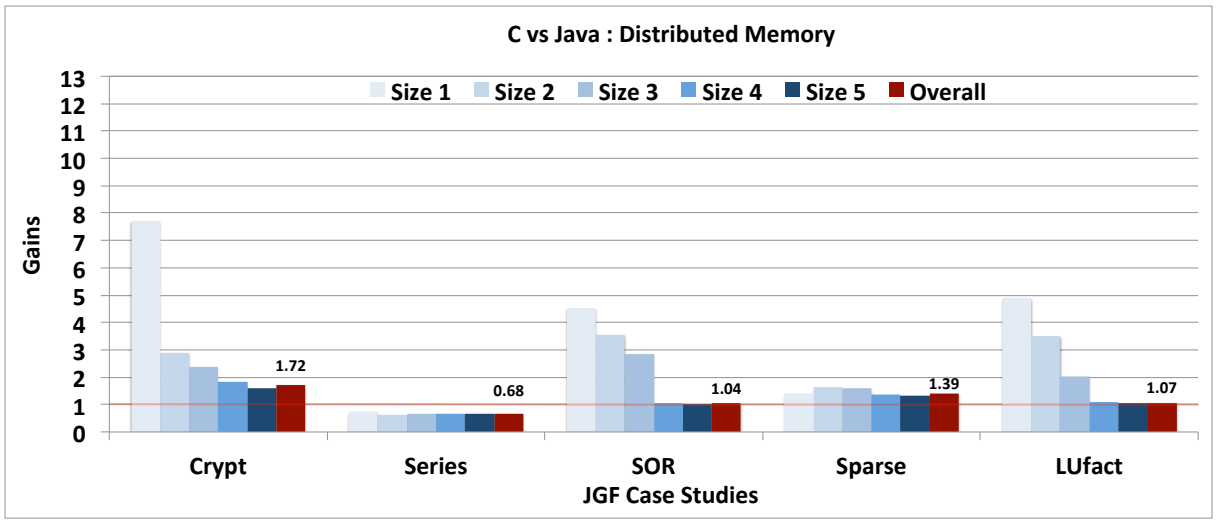


Figure 6.49: JGF - The gains of the C *vs.* Java in DM versions.

Figures 6.48 and 6.49 show the gains of C *versus* Java for their SM and DM parallel versions, respectively. As expected the parallel constructors in C had a lower overhead than those from Java, especially noticeable in the smaller inputs. For instance, in C the speedups of the SM version of Series for the smallest and biggest input are 21.20 and 22.64× and in the DM version are 22.66 and 22.81×, respectively. However, for the same case study and inputs, the speedups of the Java SM version were 8.79 and 21.02× and of the DM version 11.11 and 20.93×. That indicates, especially since Series is compute bound, that Java requires a bigger task granularity than C for the parallelism to pay off. It is also worth mentioning that in SM, C benefited more from the *first-touch* technique than Java from the flag *NUMA* (details in Table E.11). Combining the overall gains of the SM parallelizations of the Crypt, SOR, and LUFact case studies, the *first-touch* approach and flag *NUMA* provided an aggregated gain of 4.92 and 4.24, which corresponds to a reduction of 27.968 and 19.449 seconds from the execution time, respectively. Thus, the first-touch had a gain of 1.43 over the flag *NUMA*.

Initially, we expected that the gains of C over Java would be smaller in the DM than in the SM. However, that is not the case for Crypt and Sparse (the difference in Series is negligible). In Crypt the communication routine of its Java DM version was more complex and with higher overhead than the one in C. The Sparse Java DM version did not scale for two of the five inputs and the overhead of the *all reduce* routine in Java was higher than in C (results in Table F.19).

Regarding the C hybrids, we implemented one for the Series case study. However, the results of testing the hybrid and DM versions in multi-machines did not provide new useful information. The C DM and hybrid versions, running in multiple machines, are still slower than the versions with Java 9. Moreover, as with the Java intrusive implementations and with the layers of aspects, with C the hybrids also outperformed the DM version. Finally, although the DM and hybrid C versions were slower than the ones in Java 9, their speedups were higher (overall 153.93 and 172.52×, respectively). The results can be seen in more detail in Figure E.1.

6.5 Design rules

The majority of the design rules applied during the implementation of SM and DM layers were methods. By introducing methods, we are adding points that can be extended in the future, which is in line with the *design for change* principle. Indeed, several design rules added during the SM layer testing workflow were reused in the DM layers as well. Notwithstanding, to not jeopardize code quality a good balance should be achieved; otherwise, one could merely turn every statement into a method.

One problem with AspectJ is that it can lead to the introduction of empty methods [KAB07] into the base code as a way of creating artificial join points. Such methods are considered poor design [MLWR01], and therefore should be avoided. However, we did not experience these problems during the implementations of the layers, mainly because the parallelism-related constructors will typically be injected into significant chunks of code. Hence, these blocks will contain enough logic to become a method without jeopardizing the code legibility. Naturally, parallelism-related constructors that perform *around* code transformations (*e.g.*, parallel region) will be injected into blocks of code that can become a method. The problem with empty methods is more likely to happen with constructors that inject single point code transformations (*i.e.*, barriers). Most of the AOmpLib constructors have implicit barriers at the end of their execution which reduce the need to invoke barriers explicitly, while most of AOdmLib communications routines will happen before or after a *hot spot* in the base code.

In the scenarios that empty methods are needed with our framework, they would most likely also be needed in other approaches such as design patterns (*e.g.*, *decorator pattern*), inheritance, and so on. In such scenarios, the only way to avoid empty methods would be with an approach with a finer grain join point model that the one used by our framework, which would undesirably increase the fragility of such approach.

6.6 Class Extension and Decorator Pattern

In this section, we present a small study of the implementation of the MD case study with common OO strategies. We start by presenting some of the implementations details and conclude the section with a discussion about some of the advantages and disadvantages of these implementations.

We created Java intrusive implementations of the best SM and DM layers using class extension and method overriding (Figure 6.50) to compare them with our parallelism layers. The PRC are injected in the base code by extending its classes (*i.e.*, *MD* and *Particles*), overriding some of its methods, and reusing others. The subclasses of the *MD* (*i.e.*, *MD_SM* and *MD_DM*) are responsible for the management of the parallelism-related structures, (*e.g.*, thread pool), the assignment of tasks to the threads/processes, and the reduction of the appropriate variables after the force calculation. The subclasses of the *Particles* class are responsible for the parallelization of the outer loop of the force calculation.

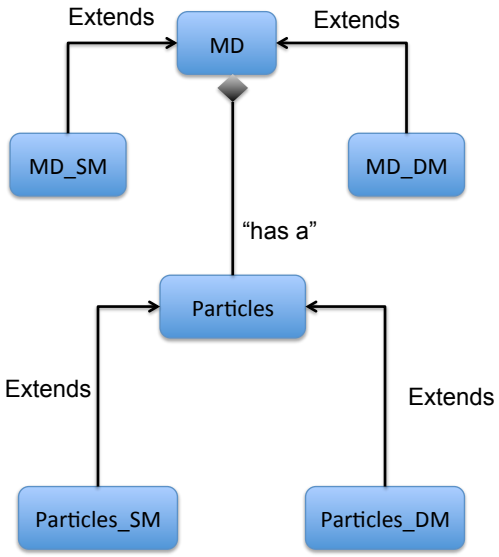


Figure 6.50: MD - Diagram of classes of using extension to implement layers.

As presented in Figure 6.51 the subclass *Particles_SM* extends the base class *Particles* (line 1), overrides the method that calculates the forces (line 4), and parallelizes the outer loop of that method (line 6). For every iteration of the parallelized loop, the method of the base class *Particles* that performs the force calculation between a particle and the remaining is invoked (line 7). Additionally, we introduced in the base code a *factory pattern* for the *MD* and *Particles* classes (Figure A.7 of appendix A) to have a flexible mechanism to change between different MD implementations (*i.e.*, sequential, SM, and DM). These implementations can be chosen at runtime based on, for instance, a value read from a configuration file. This avoids having to manually specify in the base code which classes to be used.

```
1  public class Particles_SM extends Particles {
2    ...
3    @Override
4    public void calculate_force(MD md){
5      ...
6      for( /** a given parallel for distribution **/)
7        super.forceNewtonsLaw(threadPrivateMD, i);
8      ...
9    }
10 }
```

Figure 6.51: MD - Code snippet of the *Particles_SM* class of the SM implementation.

We had to reduce some of the original constraints of the MD classes and methods, namely removing some of the *private* and *final* properties, and some of the PRC are still mixed with the domain code (*i.e.*, *factory pattern*). Nevertheless, with class extension and method overriding we were able to create the private copies of the *MD* and *Particles* object without changing the code of the force calculation methods. Moreover, the majority of the force calculation code of the base class *Particles* is reused. We achieved this by calling part of the logic of the force calculation method (*i.e.*, invoking the *forceNewtonsLaw* method). By using such a strategy, we avoided the need to apply a refactoring similar to our *for* method design rule (*i.e.*, exposing the loop range). However, a major weakness with this approach is that we are not calling the *calculate_force* method of the base class. Consequently, any change to the base class *calculate_force* method that is not inside the *forceNewtonsLaw* method will not be visible to the parallelization.

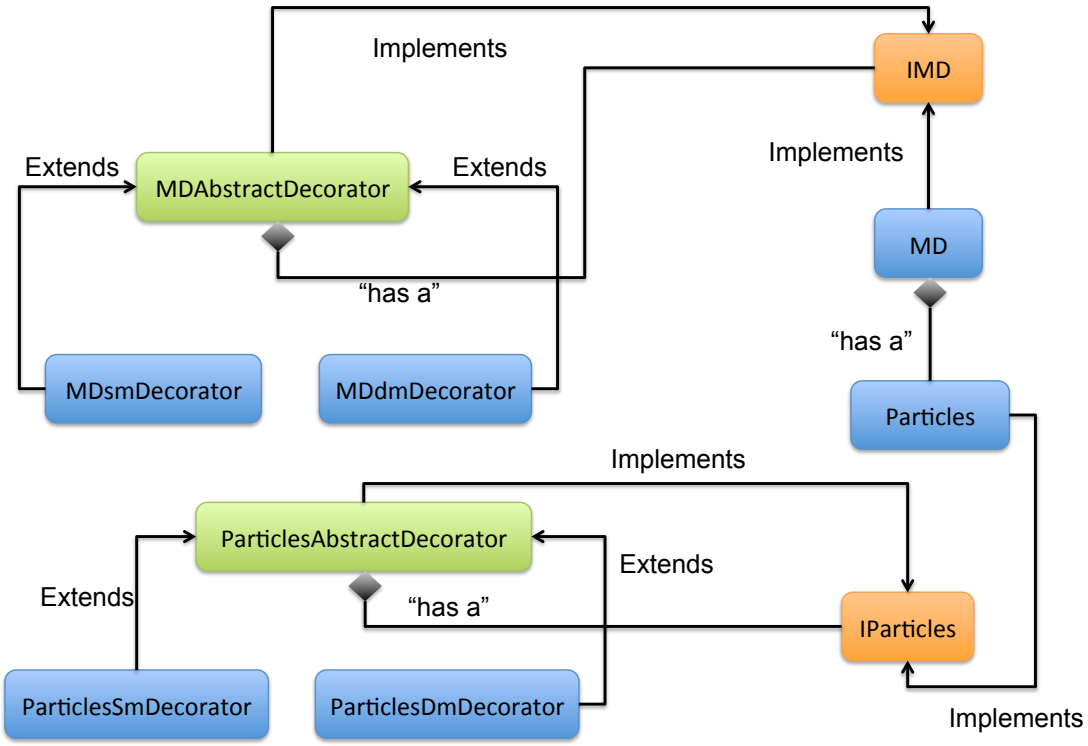


Figure 6.52: MD - Diagram of classes after the implementation of decorator pattern.

We also created a Java intrusive version based on the *decorator pattern*; Figure 6.52 shows the diagram of classes resulting from the application of that pattern. Firstly, we created a common interface (labeled "IMD" in Figure 6.52) with all the externally visible methods that the concrete implementations of the *MD* class (in our case the base *MD* sequential class) should provide. Secondly, we created an abstract decorator class ("MDAbstractDecorator") that implements the common interface and also has it as a field. Finally, we applied the same structure to the class *Particles* as well. Every time a new parallel version of the MD case study has to be implemented, it will be created as concrete classes of the abstract decorated classes (*i.e.*, *MDAbstractDecorator* and *ParticlesAbstractDecorator*). The abstract decorator classes forward all the methods of the common interfaces so that their subclasses are not obliged to overwrite them, hence minimizing code duplication. The abstract decorator class works as a mediator between its concrete classes and the correspondent base class (*e.g.*, *MD* and *Particles*).

The *decorator pattern* approach is regarded as being more maintainable and scalable than inheritance. However, compared with the inheritance approach, the decorator pattern required a significant amount of non-intrusive statements because of the extra interfaces and abstract classes. Furthermore, we also had to turn some private methods into public ones (*e.g.*, those added to the interfaces) – some of which are irrelevant to the parallelism. Another issue with this approach is that since the (abstract) decorators implement the common interfaces, all the methods of these interfaces need to be implemented, even the ones that will not be *decorated*. Consequently, leading to several one-line methods (known as *forwarding methods*) coded in the abstract decorators. Regarding complexity, those extra non-intrusive statements are negligible; however, they affect productivity. Nevertheless, if the number of concrete decorator classes increases the total of statements of the common interfaces and abstract decorator classes should remain the same.

```
1   public void runiters() // Change to public
2   {
3     for (/** Run simulation N Times **/) {
4       cicleDoMove               (...);
5       cicleForcesNewtonsLaw     (...);
6       cicleMkekin               (...);
7       cicleVelavg               (...);
8       scale_temperature         (...);
9       get_full_potential_energy (...);
10    }
11  }
```

Figure 6.53: MD - Problems with *decorator pattern*.

The biggest issue with the *decorator pattern* happened when we tried to decorate the method *cicleForcesNewtonsLaw*. Decorating the *calculate_force* method was easy because it is invoked from the exterior – this method is defined in the *Particles* class and called from the *MD* class. The class *main* has a reference to the *MD* object that is being decorated, and this *MD* object has a reference to the *Particles* object that is being decorated. However, the method *cicleForcesNewtonsLaw* is only being called internally (*i.e.*, inside the same class); when the *main* class requests a *MD* object, the factory will return the appropriate decorator, for instance, *MDsmDecorator*. When the *main* calls the method *runiters*, the *MDsmDecorator* will forward that call to the *MDAbstractDecorator*, which on its turn will forward it to the *MD* base class, leading to the execution of the code illustrated in Figure 6.53. The issue is that the method *cicleForcesNewtonsLaw* of the *MDsmDecorator* will never be called, and consequently, the method *cicleForcesNewtonsLaw* of the base class will never be decorated with the functionality of the *MDsmDecorator*. Possible solutions to deal with this issue are: - the use of the *strategy pattern* for the method *cicleForcesNewtonsLaw*; - or merely inlining the method *runiters* call in the *main* class. We have chosen the latter, and consequently, forced to make the methods from lines 4, and 6 to 9 of Figure 6.53 protected.

The issues experienced with the *decorator pattern* do not come from a bad design of the *MD* class, the *decorator pattern*, alone, is not suitable for this case. For example, the same would have happened if we tried to implement the SM layer that used *critical* regions (shown in Section 5.3.1). In this case, we would have had problems with the method *forceUpdate* of the class *Particles*. Nonetheless, we could extract the forces into a class and apply the *decorator pattern* with them as well.

We did the previous process for the Java intrusive DM layer, and in both approaches (*i.e.*, inheritance and the *decorator pattern*) we faced the same pros and cons, previously discussed. To replicate the best SM and the best DM layers as intrusive versions we created new *MD* and *Particles* classes that either extended or decorated their corresponded base classes. Each pair of new *MD* and *Particles* classes, regardless of the approach used (inheritance or *decorator pattern*), parallelizes the base code in a SM or DM fashion. For the hybrids, the goal was to reuse and combine the individual SM and DM modules[40]. However, with inheritance, this is not possible, since the SM and DM modules extend already the base classes. Thus, we created copies of the DM modules, named them hybrids, and made them extend the SM modules instead of the base classes. Building the hybrid in this way was straightforward; we were able to reuse the SM modules but duplicated the DM modules. Concerning the *decorator pattern*, the hybrid was created by returning, in the *MD* and *Particles* factories, the new objects corresponding to the DM decorators wrapping the SM decorators that in turn wrapped the (sequential) base

[40]By modules of a given environment (SM and DM), we are referring to the new *MD* and *Particles* classes created to provide the PRC concerning that environment.

classes. Hence, regarding module composition, the *decorator pattern* offered a cleaner solution than inheritance. Nonetheless, with both inheritance and *decorator pattern* approaches, even after composing the SM and DM modules, the hybrid versions were (still) not working correctly. We stumbled across the problem of properly composing the DM and SM loop parallelizations. Since in the SM and DM modules the loop parallelization is hard-coded in the loop body itself and not exposed to the API, it is not viable to compose the multiple levels of loop parallelization using conventional design patterns. For the correct loop composition among different levels of parallelization, each sub-level has to be aware of the loop transformations performed by its upper-level. Hence, in our case, the SM module needs to know the loop range transformations previously performed by the DM module. To guarantee the correct composition of the multiple loop transformations, we extracted the loop range as parameters of the force calculation method, making it part of the *Particles* API, exactly like our *for* method design rule. Therefore, the *Particles* DM class will overwrite/decorate the force calculation method, take its parameters, change them based on the DM loop distribution and then call the force calculation with the new parameters. Afterward, the *Particles* SM class will intercept that force calculation call, change its parameters accordingly and proceed to call the force calculation. Finally, at that point, the force calculation from the base (sequential) *Particles* class is called with the modified parameters resulting from the DM and SM loop transformations.

As we can see inheritance, *decorator pattern*, and other design patterns, also force the designing of the base code. Namely, creating methods to inject the additional functionality and making them public/protected. We even end-up applying the *for* method design rule to be able to compose the SM and DM modules. Inheritance offers a simple solution and enables the reuse of functionality. However, it suffers from fragile base class problem, does not scale well when composing multiple features (*e.g.*, combining SM and DM modules), *exploding class hierarchy*, among others. In the case of the parallelism, in our opinion, extending a base class with a subclass that injects parallelism creates a hierarchy that is semantically unsound. For instance, the relationship between *Particles* and *Particles_SM* has nothing to do with the domain concerns. The decorator pattern provides a flexible approach that can combine multiple features. However, it leads to a significant amount of boilerplate code, forward methods, a chain of decorators that can be hard to follow and which increases the complexity of the code. In the MD case study, the majority of the statements added were mainly concerned with the implementation of the pattern and had nothing to do with either the PRC or DRC. In both inheritance and *decorator pattern*, there are still traces of PRC in the base code, albeit significantly less compared with an intrusive approach that injects the PRC directly in the base code. Furthermore, the PRC are still scattered among the subclasses and the decorators. Hence, to reason about the parallelizations, the programmer has to go through not only multiple domain classes but also multiple parallelism-related subclasses/decorators.

Chapter 7

Conclusion and Future Work

Nowadays, high-performance clusters connect several multicore machines, leading to systems with multiple levels of parallelism. To exploit these cores software has to be developed accordingly (*i.e.*, parallelized, potentially with different paradigms). Unfortunately, the non-modular and unstructured approach of mixing the PRC and DRC, embraced by the traditional parallel programming paradigms, increases the complexity of developing parallel applications.

This thesis contributes towards the goal of reducing the complexity of PP – while achieving a competitive performance – through the concept of parallelism layers. This concept emerged from our study of AOP to develop an approach capable of reducing the complexity and increasing the structure of the development of parallel applications, with performance comparable to traditional parallel programming paradigms. This study led to several publications [MS13a, MS13b, MSS15, MSS16, MS17]. Parallelism layers help to achieve these goals by promoting the development of efficient parallel applications based on three fundamental software properties, namely modularity, pluggability, and composability. The PRC are encapsulated in modules with high cohesion and a well-defined responsibility – following the *single responsibility principle*. These modules are not tightly-coupled with the base code and can be (un)plugged from the base code without changing it. Hence, enabling the support for the base code sequential semantics. Moreover, these modules can be composed/combined with the base code to provide different parallelization strategies, which helps to find efficient parallelizations for a given environment. These properties (*i.e.*, modularity, pluggability, and composability) are tightly connected with the separation of the PRC from the DRC. This separation of concerns is achieved by the parallelism layers through simple, yet powerful, well-known OO concepts such as class extension and method overriding. By separating the PRC from the DRC, and encapsulating the PRC in proper modules that are grouped into layers, the tangling and scattering problems are avoided. The additional overhead of adding the PRC with the parallelism layers is not significant since these layers rely on low overhead mechanisms similar to class extension and method overriding.

To materialized the parallelism layers we developed an aspect-oriented framework – more specifically with AspectJ – enhanced with a methodology based on structured programming and design rules. AspectJ provided means to modularize and compose the PRC with the domain code without having to change its semantics – a fundamental characteristic of parallelism layers. AspectJ enabled us to combine the use of reusable libraries (composed with modules that tackle a specific PRC) with the injection of the constructors of these libraries into the base code in a non-invasive manner. Hence, combining the advantages of reusable libraries (*e.g.*, reusability and the ability to extend their functionality) and extending the domain functionally in a similar approach to inheritance. However, without the disadvantages of mixing the PRC and DRC, changing the structure of the domain classes, among others.

To reduce complexity and promote structure, we suggest the development of parallel applications with our framework in an incremental fashion. Programmers start by developing the code containing the DRC and afterward the layer that encapsulates the PRC. These layers are added to the domain code by plugging in the framework's aspect modules. The proposed approach is an important step to make the process of developing parallel applications more structured, faster, and cleaner. More structured because the programmer incrementally parallelizes applications by composing their base code with the parallelism-related modules, instead of adding the parallelism directly into the base code in a disorganized manner. Faster, because programmers can quickly (un)plug different parallel modules to test which one provides the best performance for a specific platform, or completely remove the PRC to reason about the base code in isolation without having to rewrite the domain code. Moreover, the framework raises the level of abstraction by providing high-level parallelism-related modules that abstract the programmer from the low-level details of PP. Cleaner because the PRC are not mixed up with the domain application code, promoting a modular and incremental software development.

Our parallelism layers rely solely on methods and objects to inject their constructors. These layers are further restricted to only public/protected non-final/static methods, which simplifies our approach, makes it more predictable and closer to how class extension works. Moreover, it helps to attenuate the pointcut fragility problem of AOP. Even though the programmer has to follow our design rules (*e.g.*, method refactoring) to inject the parallelism-related code transformations, these do not break the sequential semantics of the base code, and hence we considered that these design rules do not violate obliviousness in the context of parallelism. If we consider that the base code is oblivious of the aspect modules only when (literally) no modification should be made to it, then we argue that *true* obliviousness is impossible to achieve without a much finer grain join point model than the one provided by AspectJ. Moreover, in our opinion, such level of obliviousness would be undesirable in the context of parallelism because it would increase the fragility and complexity of developing parallel applications.

We evaluated the performance and programmability of our framework with a set of case studies executed in a cluster of multicores. The majority of the design rules introduced were methods and *for* methods; we argue that most of them made the code more readable since they introduced well-named methods with well-defined behavior. Moreover, several of them were created for the SM layer and reused in the DM layer, which highlights their importance for the parallelism-related constructors. Finally, we did not have to create empty methods [KAB07, MLWR01] to create artificial extension points. We argue that parallelism-related constructors usually are injected into blocks of code significant enough to be turned into methods. In some case studies, additional refactoring was needed, for instance, to make intrinsically sequential loops into parallelizable ones. Hence, as we can see, *true* obliviousness is hard to achieve in the context of parallelism. By making these loops parallelizable, we introduced, very subtly, PRC into the base code and made the base code aware of it. However, in our opinion, this level of obliviousness is acceptable as long as the base code sequential semantics are retained.

To implement the best SM and DM layers the intrusive Java implementations required, on average, 1.6× and 1.4× more statements than our libraries of aspects using a pointcut syntax – including the design rules applied, which account for 26% of the statements. For the parallelizations that could rely solely on annotations, our framework annotations required only half of the statements needed by the pointcuts. Although an elegant approach, annotations are not enough for more complex parallelizations that require the tuning of the parallelism-related constructors (*e.g.*, customized loop distributions and scatter/gathers). Nonetheless, our annotations can be injected into the base code non-invasively, promoting a modular approach.

For the biggest input of each case studies, the Java SM and DM intrusive implementations were, on average, 1.02× and 1.00× faster than the AOmpLib and AOdmLib, respectively. Meanwhile, for the smallest input, these values were 1.09× and 1.05×. Usually, the time difference between intrusive and our framework was higher for the smaller inputs but decreased with the increase of the input size. For the vast majority of the cases, our design rules did not meaningfully change the execution time of the sequential versions.

With our framework, we were able to improve the overall execution time of the parallelizations of 5 out of 7 JGF case studies with improvements ranging from 1.01 to 1.19×. Moreover, we were able to build hybrid parallelizations out of the best SM and DM aspect layers for 4 out of 9 used case studies. Some of these hybrids scaled up to 149.88× for 256 cores (with hyperthreading). The composition of the SM and DM layers with our framework was as easy as adding them into the build, even though some hybrid parallelizations resulted in up to 6 parallelism-related constructors injected into the same extension point.

Our design rules, the fact that our transformations are restricted to public/protected non-final/static methods, the need to make the base code intrinsically parallelizable (*e.g.*, the removal of dependencies inside a loop) can be seen as inconveniences. However, this is the price to be paid for a structured design; the time spent, in the beginning, will be compensated in the long run, with code more modular and maintainable. Parallelism layers, like other popular parallel programming paradigms, do not make *miracles*, the code has to be designed with the parallelism in mind.

From our set of design rules, the (optional) ones concerning performance are arguably from an aesthetic point of view less desirable. Regarding the *object creation* design rule, it might look strange to see a method that only returns a new object. The *set/get* performance design rule might, in the worst case scenario, led to considerable code refactoring if local variables have to be passed as parameters of several methods. In all our case studies that need private fields, we were able to use the AOmpLib *private object pointcut and annotations* approach instead of the *set/get pointcuts*. Moreover, the matrix multiplication case study was the only one that strictly needed the *object creation* design rule due to huge memory consumption in the matrix multiplication case study.

7.1 Future Work

We describe possible future work that can be done to improve the concept of parallelism layers, our methodology, and our framework.

Design rules

There is room for improvement in our data-related design rules (*e.g.*, performance design rules). Our framework could be enhanced with a mechanism to handle the data more transparently, especially in SM environments. The first step might be to force the use of setters and getters to access the data that will be injected with PRC. Adding such a restriction would reduce the expectable interface of the parallelism layers to only methods, and consequently, make our approach even simpler, predictable, and closer to OO class extension and method overriding philosophy. However, this restriction would lead to additional code refactoring. Nonetheless, most IDEs automatically introduce setters/getters methods and even replace all the direct accesses to the variables by calls to these methods. Moreover, performance-wise such methods are (likely) inlined. This approach would facilitate any potential future automation concerning our framework (*e.g.*, automatically creating pointcuts/annotations based on the method name).

Unfortunately, it is not clear yet how forcing the use of setters and getters would help to avoid our performance design rules. Another solution is to provide our own data objects to the users, or a combination of our framework with a data-centric framework, which would facilitate handling the accesses to the data. Moreover, this solution would also facilitate potential data-related optimizations (*e.g.*, tiling). We would also include the use of iterators in the loops to be parallelized, which would avoid having to expose the loop body. Notwithstanding, a proper study of the impact of those solutions on performance and programmability would have to be conducted before.

Correctness

To help with the correctness of the parallelizations, we can provide layers similar to parallelism layers but, instead, directed to ensure that there are no combinations of constructors that might lead to problems (*e.g.*, a *barrier* called inside a *master* constructor). Such *correctness layer* will likely rely heavily on the AspectJ *cflow* and *cflowbelow* pointcuts. Hence, this layer would only be turned on to check the soundness of the parallelization and turned off during the execution of the actual parallelism layer. The programmer would develop a *normal* parallelism layer with the constructors to be injected and so on. Afterward, we would provide a transparent mechanism that would allow changing the layer from parallelism to *correctness*, which would avoid having to duplicate layers.

Extending framework functionality

We plan to extend our framework to complement other hardware environments; some initial work was already done in this direction with extensions for GRID [MSS15] and GPUs [MSS16]. Unfortunately, due to having to depend on third-party tools to execute the code in those environments, additional work will be necessary to fully integrate these extensions with our framework, methodology, and design rules. Possibly with the development of intermediate software to make the bridge between the different extensions. Furthermore, we might also consider integrating some in-house aspect-oriented tools that are helpful in the context of parallelism, namely checkpoint [MS11] and profiling (*e.g.*, *PAPI*).

Automation

Most of the process of creating the parallelism-related modules follow a fixed order; creating the aspects, choosing the pointcuts, overriding methods to customize the constructors and so on. We want to develop tools (*e.g.*, a plugin) that can automatize part of this process. One possibility is to integrate such a tool with current IDE refactoring tools, for instance, most of the times a *for* method is a *method refactoring* followed by three extract *method parameters*. Moreover, such a tool could automatize part of the process of creating concrete modules and pointcuts. For instance, the user could select a block of code to apply method refactoring, and the tool in addition to applying the method refactoring would request the names of the constructor to use (*e.g.*, parallel region), the concrete module, and the layer where that constructor should be defined. Since the tool performs the *method refactoring* it has access to all the context necessary to fill up the pointcut/annotation. Since our framework only relies on the pointcut *call* to intercept method calls, the development of such a tool is easier. This process can be facilitated even further if we ended up restricting the parallelism-layer to only method calls.

Validation

From the validation point of view, it would be desirable to validate parallelism layers with large projects, and potentially using different software quality metrics. It would be desirable to understand how having the PRC centralized in a layer affects the way programmers reason about the parallelizations compared with having it spread across different classes. Moreover, it would be interesting to study for different project sizes what is the most appropriate layer granularity. *Should we have a layer per package? per parallel region ?*. Finally, it would also add additional value to implement the concept of parallelism layers using different technologies besides AspectJ and Java. There might be a better solution than AOP to implement this concept, probably one that would be simpler than AOP (in the sense that it does not have as many features as AOP) but robust enough to implement the concept of parallelism layers.

Appendix A

MD : Code and Explanations

```
1  ...
2  Particles(final int size){
3    fx = create_fx(size);
4    ...
5  }
6  public double[] create_fx(int size) {return new double [size];}
```

Figure A.1: MD - Applying the *object creation* design rule.

```
1  public class Particles {
2    ...
3    private final ReentrantLock locks[]; // Statement 1
4    ...
5    Particles(final int size)
6    {
7      locks = new ReentrantLock[size];  // Statement 2
8      for(int i = 0; i < size; i++)     // Statement 3
9        locks[i] = new ReentrantLock(); // Statement 4
10   }
11 }
```

Figure A.2: MD - Creation of an array of locks using an OpenMP/Intrusive approach.

Explanation of the total of statements used in the different implementations of the lock approach:

In the Java and OpenMP implementations, the approach of locking *per* particle would require about 14 and 11 new statements, respectively. In a possible intrusive implementation, we could declare the array of locks as a field of *Particles* and initialize it in the constructor of that class (shown in Figure A.2). Afterward, the logic to acquire/release the lock associated with a particle would be placed around the code that updates the particles' forces, adding at least 10 new statements (shown in Figure A.4).
Our AOmpLib only requires the use of 4 statements (*i.e.*, declaration of the inner aspect, the two pointcuts and the argument to be intercepted) and two additional ones to apply the *object creation design rule*. To extract the index to be used in the array of locks we reused the *forceUpdate* method, resulting in no additional design rule.

Figure A.3: MD - The explanation of the total statements needed for the lock *per* particle approach.

3

```
26  ...
27        /** Calculating thirds Newton's law */
28        try                          // Statement 1
29        {
30          locks[pB].lock();          // Statement 2
31          fx[pB] -= tmpFx;
32          fy[pB] -= tmpFy;
33          fz[pB] -= tmpFz;
34          ... // update local epot, vir interactions
35        }
36        catch(..){}                  // Statement 3
37        finally                      // Statement 4
38        {
39          unlock[pB].unlock();       // Statement 5
40        }
41      }
42    }
43    /** Update of the force of Particle A **/
44    try                              // Statement 6
45    {
46      locks[pA].lock();              // Statement 7
47      fx[pA] += fxAcc;
48      fy[pA] += fyAcc;
49      fz[pA] += fzAcc;
50    }
51    catch(..){}                      // Statement 8
52    finally                          // Statement 9
53    {
54      unlock[pA].unlock();           // Statement 10
55    }
56  ....
```

Figure A.4: MD - A possible implementation of the intrusive lock approach.

```
1     ...
2     final double fx[] = this.fx; // <- Performance design rule fx
3     final double fy[] = this.fy; // <- Performance design rule fy
4     final double fz[] = this.fz; // <- Performance design rule fz
5
6     for (int pB = pA + 1; pB < totalParticles; pB++){
7       ...
8       if(/** pB inside the radius of pA**/){
9       ...
10        forceUpdate(pB, -tmpFx, -tmpFy, -tmpFz, fx, fy, fz);
11      }
12    }
13    /** Update of the force of Particle A **/
14    forceUpdate(pA, fxAcc, fyAcc, fzAcc, fx, fy, fz);
15    /** MD control variables actualization */
16    md.updateControlVar(epot, vir, interactions);
17  }
```

Figure A.5: MD - Application of the *set/get* performance design rule.

Explanation of the total of statements among the different versions for the *set/get approach*:

The AOmpLib needed one inner aspect, 10 pointcuts and 3 *set/get* performance design rule for a total of 18 statements. The majority of the new statements from the intrusive version came from the passing the thread ID around, the creation and initialization of the new thread-related variables, and the replacement of the *problematic variables* by the new variables.

Figure A.6: MD - The explanation of the total statements needed for the *set/get* approach.

```
1  public final class MDFactory{
2    ...
3    public MD getMDversion(...)
4    {
5      if(/** SEQ version **/) return new MD(..); // MD base class
6      else if (/** SM version **/) return new MD_SM(..);
7      else if ...
8    }
9  }
```

Figure A.7: MD - Example of the *MDFactory* class.

Explanation of the total of statements among the different versions for the best SM Layer:

To implement the best SM layer composed by a parallel region with a dynamic *for* distribution and copy/reduction of the private objects, using pointcuts only, the AOmpLib used a total of 5 aspects, 6 pointcuts, 3 method definitions (to provide the object copies and the *no_wait()* clause) and 1 design rule (*for* method). In total, for the best layer, with AOmpLib we wrote 29 statements, 18 of them used to provide the copies of the *Particles* and *MD* objects. With OpenMP C 36 statements were written with only 6 of them being actual annotations (parallel region, dynamic *for* and *declare reduction*). Concerning the Java intrusive version, even with the use of extension and overriding mechanisms to minimize the number of modifications to the base code, at least 36 statements had to be written/modified. These modifications include removing *final* clauses, changing from private methods to non-private ones, creating setters/getters to manage private fields from the outside and so on. In addition to the intrusive statements we also wrote 80 non-intrusive statements spread across three different classes (*MDParallel*, *ParallelForce* and *ThreadTeam*)[a]. Since one of the new classes (*ThreadTeam*) can be reused in another case studies besides the MD, from the 80 non-intrusive statements, 19 out of them can be deducted for a total of 97[b] intrusive and non-intrusive statements. The Java intrusive version needed slightly over 3 times more statements that the layer implemented with the AOmpLib. This is not surprising since since the AOmpLib includes all the logic of the parallel region, of the copy and reduction of the private objects and so on.

[a] We excluded the *MDFactory* class from the calculations.

[b] 80 non-intrusive statements minus 19 from those that can be reused plus 36 intrusive statements.

Figure A.8: MD - The explanation of the total statements needed for the best SM Layer.

Appendix B

MD : Results

Table B.1: MD - Input sizes.

Number of particles	Size (MB)
2048	0.14
8788	0.60
19652	1.35
256k	17.17
500k	34.33

Table B.2: MD - Execution times of data dependency strategies with AOmpLib.

Number of particles	Critical	Locks	Private objects	Set/Get pointcuts
2048	0.591	**0.312**	0.371	0.426
8788	8.796	2.263	**1.623**	1.670
19652	37.834	10.924	5.870	**5.724**
250k	109.768	25.656	**13.286**	13.306
500k	395.100	88.936	50.020	**48.913**
Total	552.088	128.090	71.170	**70.039**

- The values in bold marks in each row the strategy with the lowest execution time.

Table B.3: MD - Scalability of the data dependency strategies with AOmpLib.

Version	Input	Total of threads								
		2	3	4	6	8	12	16	24	32
Critical	2048	1.08	1.43	**1.57**	1.10	1.15	1.06	1.07	1.04	1.03
	8788	1.17	1.58	**1.69**	1.34	1.29	1.29	1.31	1.27	1.22
	19652	1.21	1.55	**1.88**	1.52	1.42	1.28	1.40	1.39	1.36
	250k	1.14	1.60	1.86	**1.89**	1.55	1.49	1.47	1.41	1.44
	500k	1.13	1.59	1.87	**1.96**	1.60	1.51	1.55	1.41	1.48
Locks	2048	1.02	1.40	1.80	2.21	2.31	2.89	**2.97**	2.90	2.67
	8788	1.11	1.74	2.23	3.46	4.55	6.00	5.22	6.43	**6.56**
	19652	1.18	1.91	2.31	3.51	4.71	**6.49**	5.23	5.13	6.05
	250k	1.11	1.86	2.55	3.74	4.65	7.01	**8.07**	7.42	7.19
	500k	1.13	1.75	2.29	3.67	4.92	7.31	**8.72**	8.37	7.73
Private objects	2048	1.55	2.03	2.39	**2.50**	2.49	2.29	2.15	1.76	1.57
	8788	1.97	2.99	3.69	5.55	6.77	8.19	7.30	**9.14**	8.40
	19652	2.05	3.12	4.02	5.38	7.22	10.20	9.06	**12.09**	11.98
	250k	2.00	2.99	3.60	5.63	7.76	11.39	12.67	13.49	**15.59**
	500k	2.00	3.00	3.98	5.68	7.58	11.73	13.27	14.18	**15.51**
S/G pointcuts	2048	1.47	1.85	**2.18**	1.62	1.58	1.58	1.07	0.90	1.37
	8788	1.98	2.87	3.76	5.34	6.56	7.11	7.07	**8.89**	8.83
	19652	2.09	2.96	3.86	5.52	7.17	9.72	8.75	11.22	**12.39**
	250k	1.99	3.00	3.61	5.91	7.86	11.55	12.94	13.11	**15.56**
	500k	1.96	2.94	3.92	5.97	7.69	11.48	13.93	14.21	**15.86**

- The values in bold represent the ones with the greatest speedup;
- The speedups are calculated using the lowest execution time of the sequential code base with or without the design rules.

Table B.4: MD - Max speedups of the data dependency strategies with AOmpLib.

Number of particles	Critical	Locks	Private objects	Set/Get pointcuts
2048	1.57	**2.97**	2.50	2.18
8788	1.69	6.56	**9.14**	8.89
19652	1.88	6.49	12.09	**12.39**
250k	1.89	8.07	**15.59**	15.56
500k	1.96	8.72	15.51	**15.86**
Overall	1.94	8.35	15.03	**15.27**

- The values in bold represent the ones with the greatest speedup;
- The speedups are calculated using the lowest execution time of the sequential code base with or without the design rules.

Table B.5: MD - Execution times of load balancing strategies with AOmpLib/AOdmLib.

	SM			DM		
# of particles	Static (chunk=1)	Dynamic (chunk=1)	Manual	Static (chunk=1)	Dynamic (chunk=1)	Manual
2048	0.371	**0.330**	0.371	**0.153**	0.258	0.157
8788	1.623	**1.263**	1.628	**1.036**	1.504	1.038
19652	5.870	**4.379**	5.862	4.235	4.755	**4.231**
250k	13.286	**12.316**	13.464	13.538	**13.491**	13.550
500k	50.020	**46.732**	49.425	49.260	**48.892**	49.298
Total	71.170	**65.020**	70.750	**68.222**	68.900	68.274

- The values in bold represent the ones with the lowest execution time.

Table B.6: MD - Speedups of the load balancing strategies with AOmpLib/AOdmLib.

	SM			DM		
# of particles	Static (chunk=1)	Dynamic (chunk=1)	Manual	Static (chunk=1)	Dynamic (chunk=1)	Manual
2048	2.50	**2.81**	2.50	**6.06**	3.59	5.90
8788	9.14	**11.75**	9.11	**14.32**	9.87	14.30
19652	12.09	**16.20**	12.10	16.75	14.92	**16.77**
250k	15.59	**16.81**	15.38	15.30	**15.35**	15.28
500k	15.51	**16.60**	15.70	15.75	**15.87**	15.74
Overall	15.03	**16.45**	15.12	**15.68**	15.52	15.67

- The values in bold represent the ones with the greatest speedup;
- The speedups are calculated using the lowest execution time of the sequential execution with or without the design rules.

Table B.7: MD - Execution times of the different sequential versions.

Number of particles	C	Code Base	Design Rules
2048	0.880	0.927	0.927
8788	14.707	14.881	14.838
19652	69.424	71.087	70.946
250k	208.081	207.178	207.077
500k	777.449	776.431	775.811
Total	1070.541	1070.504	1069.599

- Neither the C nor the Base Code includes the design rules.

Table B.8: MD - Comparison gains between different sequential versions.

Number of particles	C *vs.* Java	Base Code *vs.* Design Rules
2048	1.05	1.00
8788	1.01	1.00
19652	1.02	1.00
250k	1.00	1.00
500k	1.00	1.00
Overall	1.00	1.00

- Java = The fastest between Base Code and Design Rules;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table B.9: MD - Execution times of the best SM/DM versions.

	SM			DM		
Number of Particles	C	Java	Aspects	C	Java	Aspects
2048	0.064	0.248	0.330	0.070	0.145	0.153
8788	0.877	1.165	1.263	0.823	1.032	1.036
19652	4.045	4.175	4.379	3.897	4.213	4.235
250k	12.147	12.106	12.316	13.131	13.311	13.538
500k	45.660	46.680	46.732	49.388	49.135	49.260
Total	62.793	64.374	65.020	67.309	67.836	68.222

Table B.10: MD - Speedups of the best SM/DM versions.

	SM				DM			
# of Particles	C	C adjust	Java	Aspects	C	C adjust	Java	Aspects
2048	13.75	14.48	3.74	2.81	12.57	13.24	6.39	6.06
8788	16.77	16.92	12.74	11.75	17.87	18.03	14.38	14.32
19652	17.16	17.54	16.99	16.20	17.81	18.21	16.84	16.75
250k	17.13	17.05	17.11	16.81	15.85	15.77	15.56	15.26
500k	17.03	16.99	16.62	16.60	15.74	15.71	15.79	15.75
Overall	17.05	17.03	16.62	16.45	15.90	15.89	15.77	15.68

- The speedups of Java, Aspects, and C adjust are calculated using the lowest execution time of the sequential code base with or without the design rules;
- Speedups of C are base on the execution time of its sequential version.

Table B.11: MD - Comparison gains between different versions.

		SM		DM	
		Gains	Time Difference (s)	Gains	Time Difference (s)
Java *vs.* Aspects	2048	1.33	0.082	1.06	0.008
	8788	1.08	0.098	1.00	0.004
	19652	1.05	0.204	1.01	0.022
	250k	1.02	0.210	1.02	0.227
	500k	1.00	0.052	1.00	0.125
	Overall	**1.01**	0.646	**1.01**	0.386
C *vs.* Java	2048	3.88	0.184	2.07	0.075
	8788	1.33	0.288	1.25	0.209
	19652	1.03	0.130	1.08	0.316
	250k	1.00	-0.041	1.01	0.180
	500k	1.02	1.020	0.99	-0.253
	Overall	**1.03**	1.581	**1.01**	0.527

- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table B.12: MD - Execution times of the DM/Hybrid with 8 machines.

# of Particles	C		Java		Aspects	
	DM	Hybrid	DM	Hybrid	DM	Hybrid
19652	1.212	0.782	1.315	1.110	1.340	1.189
250K	1.850	1.657	1.971	1.947	2.000	1.982
500K	6.475	5.914	6.686	6.536	6.761	6.586
Total	**9.537**	**8.353**	**9.972**	**9.593**	**10.101**	**9.757**

Table B.13: MD - Speedups of the DM and Hybrid with 8 machines.

Number of Particles	C		C adjust		Java		Aspects	
	DM	Hybrid	DM	Hybrid	DM	Hybrid	DM	Hybrid
19652	57.28	88.78	58.54	90.72	53.95	63.92	52.94	59.67
250K	112.48	125.58	111.93	124.97	105.06	106.36	103.54	104.48
500K	120.07	131.46	119.82	131.18	116.04	118.70	114.75	117.80
Overall	110.62	126.30	110.50	126.16	105.68	109.85	104.33	108.01

- The speedups of Java, Aspects, and C adjust are calculated using the lowest execution time of the sequential code base with or without the design rules;
- Speedups of C are base on the execution time of its sequential version.

Table B.14: MD - Comparison gains between versions for the DM/Hybrid with 8 machines.

	# of Particles	DM		Hybrid	
		Gains	Time Difference (s)	Gains	Time Difference (s)
Java *vs.* Aspects	19652	1.02	0.025	1.07	0.079
	250K	1.01	0.029	1.02	0.035
	500K	1.01	0.075	1.01	0.050
	Overall	**1.01**	0.129	**1.02**	0.164
C *vs.* Java	19652	1.08	0.103	1.42	0.328
	250K	1.07	0.121	1.18	0.290
	500K	1.03	0.211	1.11	0.622
	Overall	**1.05**	0.435	**1.15**	1.240

- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table B.15: MD - Gains of the Hybrid *vs.* DM with 8 machines.

		Hybrid *vs.* DM	
	# of	Gains	Time Difference (s)
C	19652	1.55	0.430
	250K	1.12	0.193
	500K	1.09	0.561
	Overall	**1.14**	1.184
Java	19652	1.18	0.205
	250K	1.01	0.024
	500K	1.02	0.150
	Overall	**1.04**	0.379
Aspects	19652	1.13	0.151
	250K	1.01	0.018
	500K	1.03	0.175
	Overall	**1.04**	0.344

- Time Difference Hybrid *vs.* DM = execution time of DM - execution time of Hybrid;
- Gains Hybrid *vs.* DM = $\frac{\texttt{execution time of DM}}{\texttt{execution time of Hybrid}}$.

Table B.16: MD - Comparison between new and old Hybrid versions with 8 machines.

	# of Particles	New Hybrid		Old Hybrid	
		Gains	Time Difference (s)	Gains	Time Difference (s)
Java *vs.* Aspects	19652	1.11	0.122	1.07	0.079
	250K	1.10	0.180	1.02	0.035
	500K	1.09	0.556	1.01	0.050
	Overall	**1.10**	0.858	**1.02**	0.164
C *vs.* Java	19652	1.36	0.285	1.42	0.328
	250K	1.09	0.145	1.18	0.290
	500K	1.02	0.116	1.11	0.622
	Overall	**1.07**	0.546	**1.15**	1.240

- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table B.17: MD - Gains of new and old Hybrid *vs.* DM with 8 machines.

		Hybrid *vs.* DM			
		New Hybrid		Old Hybrid	
	# of	Gains	Time Difference (s)	Gains	Time Difference (s)
Java	19652	1.23	0.248	1.18	0.205
	250K	1.09	0.169	1.01	0.024
	500K	1.11	0.656	1.02	0.150
	Overall	**1.12**	1.073	**1.04**	0.379

- Time Difference Hybrid *vs.* DM = execution time of DM - execution time of Hybrid;
- Gains Hybrid *vs.* DM = $\frac{\texttt{execution time of DM}}{\texttt{execution time of Hybrid}}$.

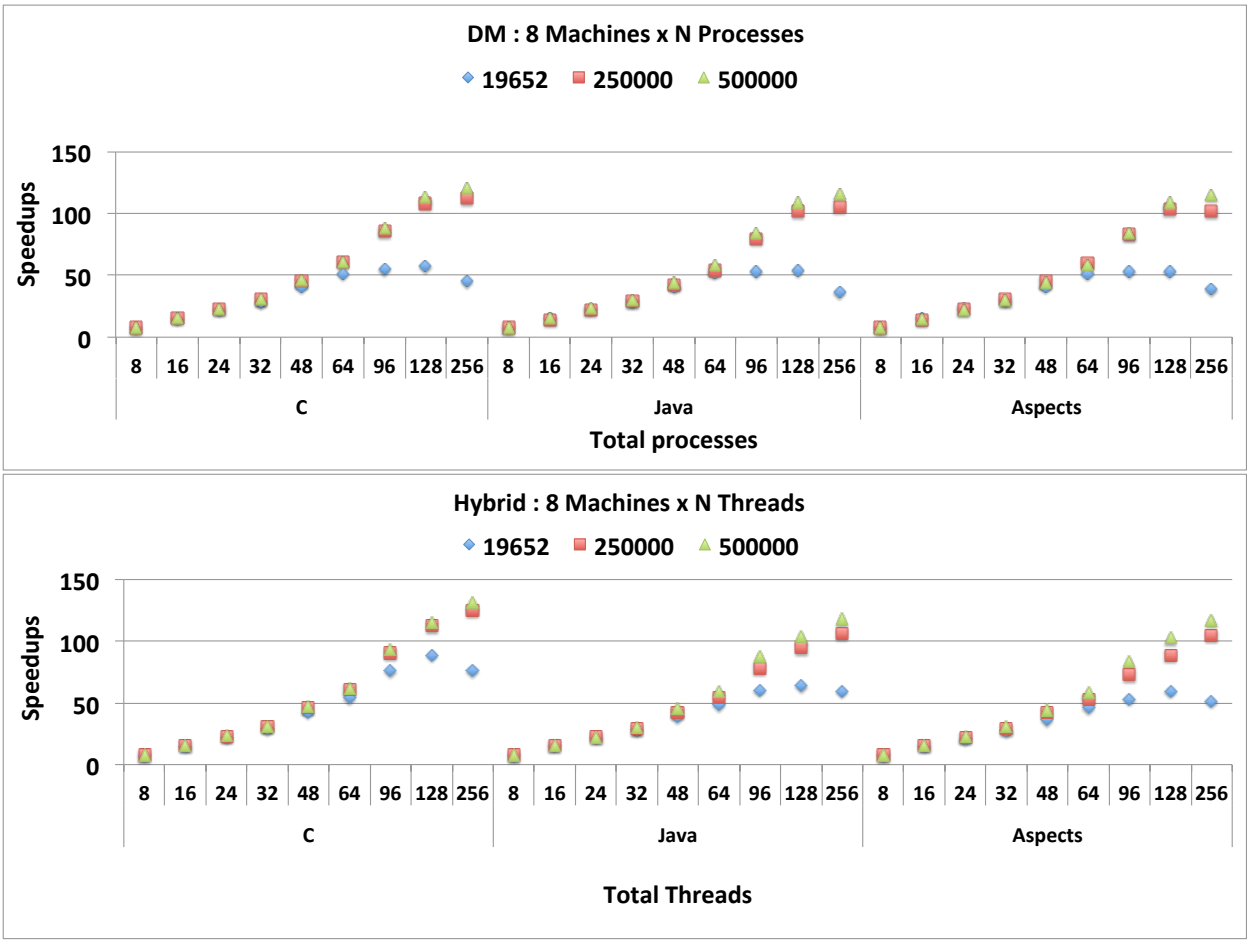


Figure B.1: MD - Scalability of the DM/Hybrid with 8 machines.

Appendix C

MM : Results

Table C.1: MM - Input sizes.

Matrices sizes	Size (MB)
1024^2	24
2048^2	96
4096^2	384
8192^2	1536
16384^2	6144

Table C.2: MM - Total of iterations.

Matrices sizes	Outer loop	Inner loop
1024^2	4	32
2048^2	8	64
4096^2	16	128
8192^2	32	256
16384^2	64	512

- Outer loop iterations = $\frac{\texttt{Total columns of matrix C}}{\texttt{Total columns of the tile}}$.
- Inner loop iterations = $\frac{\texttt{Total lines of matrix C}}{\texttt{Total lines of the tile}}$.

Table C.3: MM - Execution times of the sequential versions.

		Java		Design Rules	
Matrix Size	C	No Flags	Flags	No Flags	Flags
1024^2	0.264	0.336	0.241	0.337	0.240
2048^2	2.137	2.379	1.426	2.432	1.427
4096^2	17.076	18.842	10.819	18.667	10.810
8192^2	142.857	155.185	95.646	156.732	95.746
16384^2	1239.721	1341.674	853.209	1346.559	852.883
Total	1402.055	1518.416	961.341	1524.727	961.106

Table C.4: MM - Comparison gains between different sequential versions.

	C *vs.* Java		Java (Flags) *vs.*	(Java) Flags *vs.*
Matrix Size	No Flags	Flags	Design Rules (Flags)	(Java) No Flags
1024^2	1.27	0.91	1.00	1.39
2048^2	1.11	0.67	1.00	1.67
4096^2	1.10	0.63	1.00	1.74
8192^2	1.09	0.67	1.00	1.62
16384^2	1.08	0.69	1.00	1.57
Overall	1.08	0.69	1.00	1.58

- The flags were not applied to the C versions;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table C.5: MM - Execution times of the different versions.

	SM				DM		
Matrix Size	C	Java	Aspects	JOMP	C	Java	Aspects
1024^2	0.021	0.123	0.125	0.138	0.047	0.166	0.169
2048^2	0.151	0.288	0.266	0.322	0.278	0.462	0.491
4096^2	1.164	0.991	0.994	1.073	1.715	1.551	1.599
8192^2	9.131	6.227	6.496	6.467	11.595	8.704	8.896
16384^2	71.423	47.938	49.505	49.400	84.199	56.732	56.496
Total	81.890	55.567	57.386	57.400	97.834	67.615	67.651

- The execution time of the Java, Aspects, and JOMP implementations are the best execution time of these versions with and without the JVM performance Flags;
- For the 16384^2 input size, the tests of the DM version were limited to 16 processes do due memory constrains.

Table C.6: MM - Time spent during communication over 8 machines.

Matrices	Java	C
8192^2	3.808	2.066
16384^2	13.979	8.560

Table C.7: MM - Comparison gains between different versions.

		SM		DM	
		Gains	Time Difference (s)	Gains	Time Difference (s)
(Java) Flags *vs.* (Java) No Flags	1024^2	1.14	0.017	0.81	-0.040
	2048^2	1.26	0.074	0.97	-0.015
	4096^2	1.45	0.448	1.30	0.467
	8192^2	1.50	3.126	1.47	4.096
	16384^2	1.53	25.203	1.62	34.926
	Overall	**1.52**	28.868	**1.58**	39.434
Java *vs.* Aspects (Flags)	1024^2	1.02	0.002	1.02	0.003
	2048^2	0.92	-0.022	1.06	0.029
	4096^2	1.00	0.003	1.03	0.048
	8192^2	1.04	0.269	1.02	0.192
	16384^2	1.03	1.567	1.00	-0.236
	Overall	**1.03**	1.819	**1.00**	0.036
C *vs.* Java (Flags)	1024^2	5.86	0.102	3.53	0.119
	2048^2	1.91	0.137	1.66	0.184
	4096^2	0.85	-0.173	0.90	-0.164
	8192^2	0.68	-2.904	0.75	-2.891
	16384^2	0.67	-23.485	0.67	-27.467
	Overall	**0.68**	-26.323	**0.69**	-30.219
C *vs.* Java (No Flags)	1024^2	6.67	0.119	3.53	0.119
	2048^2	2.40	0.211	1.66	0.184
	4096^2	1.24	0.275	1.18	0.303
	8192^2	1.02	0.222	1.10	1.205
	16384^2	1.02	1.718	1.09	7.459
	Overall	**1.03**	2.545	**1.09**	9.270
Aspects *vs.* JOMP (Flags)	1024^2	1.10	0.013	-	-
	2048^2	1.21	0.056	-	-
	4096^2	1.08	0.079	-	-
	8192^2	1.00	-0.029	-	-
	16384^2	1.00	-0.105	-	-
	Overall	**1.00**	0.014	-	-

- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table C.8: MM - Speedups of the different SM versions.

Matrix Size	C	C Adjust	Java	Aspects	JOMP
1024^2	12.57	11.43	1.95	1.92	1.74
2048^2	14.15	9.44	4.95	5.36	4.43
4096^2	14.67	9.29	10.91	10.88	10.07
8192^2	15.65	10.47	15.36	14.72	14.79
16384^2	17.36	11.94	17.79	17.23	17.26
Overall	17.12	11.74	17.29	16.75	16.74

- The speedups of Java, Aspects, and C adjust are calculated using the lowest execution time of the sequential code base (with flags) with or without the design rules;
- Speedups of C are base on the execution time of its sequential version.

Table C.9: MM - Speedups of the different DM versions.

Matrix Size	C	C Adjust	Java	Aspects
1024^2	5.62	5.11	1.45	1.42
2048^2	7.69	5.13	3.09	2.90
4096^2	9.96	6.30	6.97	6.76
8192^2	12.32	8.25	10.99	10.75
16384^2	14.72	10.13	15.03	15.10
Overall	14.33	9.82	14.21	14.21

- The speedups of Java, Aspects, and C adjust are calculated using the lowest execution time of the sequential code base (with flags) with or without the design rules;
- Speedups of C are base on the execution time of its sequential version.

Table C.10: MM - Execution times of the DM and Hybrid with 8 machines.

Matrices Size	C		Java		Aspects	
	DM	Hybrid	DM	Hybrid	DM	Hybrid
8192^2	5.280	3.694	8.475	4.854	8.493	5.235
16384^2	31.372	20.045	35.441	20.674	35.210	20.732
Total	36.652	23.739	43.916	25.528	43.703	25.967

Table C.11: MM - Speedups of the DM and Hybrid with 8 machines.

Matrices Size	C		C adjust		Java		Aspects	
	DM	Hybrid	DM	Hybrid	DM	Hybrid	DM	Hybrid
8192^2	27.06	38.67	18.11	25.89	11.29	19.70	11.26	18.27
16384^2	39.52	61.85	27.19	42.55	24.06	41.25	24.22	41.14
Overall	37.72	58.24	25.88	39.96	21.60	37.16	21.70	36.53

Table C.12: MM - Comparison of the DM and Hybrid with 8 machines.

		DM		Hybrid	
		Gains	Time Difference (s)	Gains	Time Difference (s)
(Java) Flags	8192^2	1.14	1.205	1.11	0.539
vs.	16384^2	1.25	8.869	1.16	3.385
(Java) No Flags	**Overall**	**1.23**	10.074	**1.15**	3.924
Java	8192^2	1.00	0.018	1.08	0.381
vs.	16384^2	0.99	-0.231	1.00	0.058
Aspects	**Overall**	**1.00**	-0.213	**1.02**	0.439
C	8192^2	1.61	3.195	1.31	1.160
vs.	16384^2	1.13	4.069	1.03	0.629
Java	**Overall**	**1.20**	7.264	**1.08**	1.789

- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table C.13: MM - Gains of the Hybrid *vs.* DM with 8 machines.

		Hybrid *vs.* DM	
		Gains	Time Difference (s)
C	8192^2	1.43	1.586
	16384^2	1.57	11.327
	Overall	**1.54**	12.913
Java	8192^2	1.75	3.621
	16384^2	1.71	14.767
	Overall	**1.72**	18.388
Aspects	8192^2	1.62	3.258
	16384^2	1.70	14.478
	Overall	**1.68**	17.736

- Time Difference Hybrid *vs.* DM = execution time of DM - execution time of Hybrid;
- Gains Hybrid *vs.* DM = $\frac{\texttt{execution time of DM}}{\texttt{execution time of Hybrid}}$.

Appendix D

MM : Code and Figures

```
1   public void packingCacheL3(int jj, double bb[][])
2   {
3     for(int k = 0; k < maxRowB; k += 4)// Loads 4 lines
4     {
5       for (int j = 0; j < tilej; j++)
6       {
7         bb[k/4][j]            = B[k][jj+j];
8         bb[k/4][j+tilej]       = B[k+1][jj+j];
9         bb[k/4][j+tilej * 2]   = B[k+2][jj+j];
10        bb[k/4][j+tilej * 3]   = B[k+3][jj+j];
11      }
12    }
13  }
```

Figure D.1: MM - Sequential cache L3 packing of matrix B into sub-matrix `bb`.

```
1   public void packingCacheL3(int begin, int end, int step,  int jj, double bb[][])
2   {
3     for(int k = begin; k < end; k += step)// Loads 4 lines
4     {
5       for (int j = 0; j < tilej; j++)
6       {
7         bb[k/4][j]            = B[k][jj+j];
8         bb[k/4][j+tilej]       = B[k+1][jj+j];
9         bb[k/4][j+tilej * 2]   = B[k+2][jj+j];
10        bb[k/4][j+tilej * 3]   = B[k+3][jj+j];
11      }
12    }
13  }
```

Figure D.2: MM - Application of the *for* design rule in the *packingCacheL3* method.

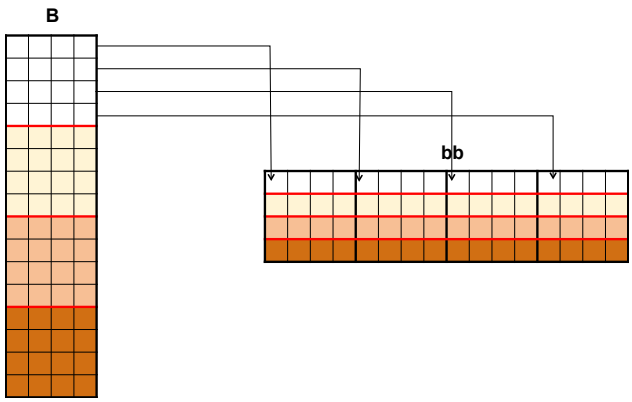


Figure D.3: MM - L3 packing of matrix B into sub-matrix bb.

```
1     public final class MM {
2       ...
3       public final parallelRegion()
4       {
5         final double bb[][] = new double[maxRowB/4][tilej*4+1];
6         for(int threadID = 1; threadID < TotalThreads; threadID++)
7         {
8           poolThreads[threadID].execute(() ->
9           {
10            matrixMultiplication(threadID, bb);
11            ...
12          });
13        }
14      }
15      matrixMultiplication(threadID, bb);
16      ...
17  }
```

Figure D.4: MM - The parallel region in the SM Java intrusive version.

```
1   public final aspect SM_MM extends SM_Layer {
2     ...
3     static aspect parallelRegion extends Sm_Parallel
4     {
5       pointcut parallel() : call(... matrixMultiplication());
6     }
7
8     static aspect sharedBB extends Sm_SharedObject
9     {
10      pointcut single():call(...createPackBB());
11    }
12
13    static aspect workSharing extends Sm_For
14    {
15      pointcut for_dynamic() :
16      (
17        call(... mmTitleMatrixC (int, int, int, ...)) ||
18        call(... packingCacheL3 (int, int, int, ...))
19      );
20    }
21  }
```

Figure D.5: MM - The best SM layer using pointcuts.

Explanation of the total number of statements of the different versions for the best SM layer:

The best SM layer with a pointcut-based approach (shown in Figure D.5) needed 4 aspects, 4 pointcuts, and 3 design rules. The design rules modified/added 8 statements and the layer of aspect another 8 (only 5 in the case of the annotation-based approach). The Java intrusive version, OpenMP C, and JOMP used 33, 3, and 10[a] statements, respectively.

[a]3 statements used in annotations and 7 for the removal of the *final* clauses.

Figure D.6: MM - The explanation of the total statements needed for the SM layer.

Explanation of the total number of statements of the different versions for the best DM layer:

In total, with the AOdmLib, we added/modified 15 statements into/from the base code as result of applying the design rules and the collateral side effects. Additionally, we needed 23 statements more to define the layer of aspects. For the Java MPI intrusive version, we wrote/modified 51 statements, of which 20 were used to write the methods to perform the scattering, gathering, and reduction of the matrices. The C MPI intrusive version needed 38 statements[a].

[a]The discrepancies between the number of statements come from the difference between C and JAVA languages.

Figure D.7: MM - The explanation of the total statements needed for the DM layer.

Appendix E

JGF Benchmark : Results

Table E.1: JGF - Number of elements of the inputs.

	Number of elements						
Input size	Crypt	Ray	Series	SOR	Sparse	MC	LUFact
Size 1	3M	150	10k	$1k^2$	250k	10k	500^2
Size 2	20M	500	100k	$1.5k^2$	500k	60k	$1k^2$
Size 3	50M	1500	1M	$2k^2$	2.5M	90k	$2k^2$
Size 4	200M	2000	2M	$10k^2$	5M	120k	$8k^2$
Size 5	900M	2500	2.5M	$15k^2$	7.5M	150k	$16k^2$

Table E.2: JGF - Input sizes.

	Size (MB)						
Input size	Crypt	Ray	Series	SOR	Sparse	MC	LUFact
Size 1	8.58	260	0.15	7.63	4.58	283	1.94
Size 2	57.22	560	1.54	17.17	9.16	975	7.73
Size 3	143.05	577	15.41	30.52	45.78	1555	30.87
Size 4	572.21	589	30.82	762.94	91.55	1765	493.31
Size 5	2574.92	606	38.52	1716.61	137.33	2222	1972.82

- The values of the RayTracer (Ray) and MC are rough measurements in the peak of the JVM memory used;
- The values of the Crypt, Series, SOR, Sparse, and LUFact measures the memory that the application needs for the input provided, however this value does not include the memory reserved for the JVM.

Table E.3: JGF - Execution times of the sequential versions.

Version	Input	Crypt	Ray	Series	SOR	Sparse	MC	LUFact
C	Size 1	0.114	-	2.629	0.167	0.234	-	0.031
	Size 2	0.765	-	26.324	0.375	0.508	-	0.233
	Size 3	1.914	-	263.170	0.808	3.862	-	2.289
	Size 4	7.656	-	526.307	22.859	8.323	-	19.596
	Size 5	34.466	-	658.066	52.731	17.710	-	37.495
	Overall	**44.915**	-	**1476.496**	**76.940**	**30.637**	-	**59.644**
JGF	Size 1	0.178	1.283	0.968	0.294	0.243	2.186	0.066
	Size 2	0.962	13.397	13.807	0.610	0.517	12.828	0.338
	Size 3	2.343	120.874	161.595	1.132	3.785	19.507	2.880
	Size 4	9.254	212.227	327.597	29.447	8.281	25.854	24.582
	Size 5	41.510	341.262	410.986	64.343	17.650	32.254	44.424
	Overall	**54.247**	**689.043**	**914.953**	**95.826**	**30.476**	**92.629**	**72.290**
JGF Improved	Size 1	-	0.945	-	0.207	-	-	-
	Size 2	-	10.033	-	0.447	-	-	-
	Size 3	-	88.617	-	0.878	-	-	-
	Size 4	-	157.986	-	22.816	-	-	-
	Size 5	-	246.515	-	53.264	-	-	-
	Overall	-	**504.096**	-	**77.612**	-	-	-
Design Rules	Size 1	0.179	0.945	0.967	0.209	0.241	2.187	0.064
	Size 2	0.962	10.045	13.809	0.452	0.516	12.814	0.338
	Size 3	2.342	88.482	161.606	0.878	3.773	19.487	2.892
	Size 4	9.245	158.397	327.648	23.015	8.260	25.851	23.390
	Size 5	41.462	245.561	410.964	53.094	17.644	32.223	42.608
	Overall	**54.190**	**503.430**	**914.994**	**77.648**	**30.434**	**92.562**	**69.292**

Table E.4: JGF - Improvements over the sequential versions.

	Ray		SOR	
	Gains	Time Difference (s)	Gains	Time Difference (s)
Size 1	1.36	0.338	1.42	0.087
Size 2	1.34	3.364	1.36	0.163
Size 3	1.36	32.257	1.29	0.254
Size 4	1.34	54.241	1.29	6.631
Size 5	1.38	94.747	1.21	11.079
Overall	**1.37**	184.947	**1.23**	18.214

- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table E.5: JGF - Fastest execution times of the SM versions.

Version	Input	Crypt	Ray	Series	SOR	Sparse	MC	LUFact
C	Size 1	0.006	-	0.124	0.016	0.014	-	0.006
	Size 2	0.040	-	1.169	0.034	0.032	-	0.028
	Size 3	0.100	-	11.632	0.060	0.280	-	0.154
	Size 4	0.394	-	23.253	4.905	0.679	-	4.755
	Size 5	1.675	-	29.065	10.939	1.634	-	8.577
	Overall	**2.215**	-	**65.243**	**15.954**	**2.639**	-	**13.520**
JGF	Size 1	0.086	0.288	0.134	0.144	0.069	0.199	0.076
	Size 2	0.158	1.281	0.851	0.193	0.096	0.786	0.208
	Size 3	0.226	10.514	8.558	0.269	0.390	1.400	0.458
	Size 4	0.582	18.354	16.971	5.472	0.825	1.749	7.256
	Size 5	2.159	28.909	21.253	12.429	1.718	2.018	10.894
	Overall	**3.211**	**59.346**	**47.767**	**18.507**	**3.098**	**6.152**	**18.892**
JGF Improved	Size 1	0.073	0.223	0.110	0.056	-	0.189	0.066
	Size 2	0.125	0.741	0.730	0.083	-	0.701	0.160
	Size 3	0.195	4.950	7.713	0.118	-	1.273	0.408
	Size 4	0.538	8.488	15.514	5.457	-	1.589	5.435
	Size 5	1.996	13.026	19.549	12.511	-	1.834	10.347
	Overall	**2.927**	**27.428**	**43.616**	**18.225**	-	**5.586**	**16.416**
Aspects (AOmpLib)	Size 1	0.079	0.233	0.126	0.058	0.074	0.188	0.073
	Size 2	0.133	0.764	0.737	0.089	0.107	0.727	0.169
	Size 3	0.212	4.941	7.768	0.126	0.390	1.284	0.470
	Size 4	0.571	8.557	15.640	5.495	0.821	1.605	5.369
	Size 5	2.098	13.064	19.538	12.556	1.733	1.859	10.419
	Overall	**3.093**	**27.559**	**43.809**	**18.324**	**3.125**	**5.663**	**16.500**
JOMP	Size 1	0.097	0.237	0.150	0.074	0.085	0.209	0.073
	Size 2	0.173	0.772	0.765	0.106	0.122	0.726	0.152
	Size 3	0.258	4.970	7.756	0.144	0.416	1.302	0.354
	Size 4	0.642	8.518	15.605	5.485	0.881	1.627	5.517
	Size 5	2.246	13.010	19.537	12.493	1.771	1.874	10.668
	Overall	**3.416**	**27.507**	**43.813**	**18.302**	**3.275**	**5.738**	**16.764**

Table E.6: JGF - Fastest execution times of the DM versions.

Version	Input	Crypt	Ray	Series	SOR	Sparse	MC	LUFact
C	Size 1	0.013	-	0.116	0.017	0.131	-	0.009
	Size 2	0.057	-	1.151	0.034	0.271	-	0.032
	Size 3	0.126	-	11.535	0.061	2.356	-	0.187
	Size 4	0.466	-	23.088	5.154	6.007	-	5.339
	Size 5	2.068	-	28.848	11.517	9.570	-	10.905
	Overall	**2.730**	-	**64.738**	**16.783**	**18.335**	-	**16.472**
JGF	Size 1	0.100	0.190	0.087	0.124	0.192	0.945	0.059
	Size 2	0.165	1.048	0.833	0.231	0.460	3.885	0.139
	Size 3	0.299	7.071	7.953	0.295	4.016	5.345	0.392
	Size 4	0.854	12.527	15.888	5.426	8.837	6.504	5.708
	Size 5	3.277	19.271	19.700	11.829	13.492	9.145	11.371
	Overall	**4.695**	**40.107**	**44.461**	**17.905**	**26.997**	**25.824**	**17.669**
JGF Improved	Size 1	-	0.144	0.087	0.077	0.185	-	0.044
	Size 2	-	0.760	0.740	0.121	0.445	-	0.112
	Size 3	-	4.958	7.810	0.173	3.792	-	0.380
	Size 4	-	8.683	15.668	5.377	8.408	-	5.731
	Size 5	-	13.416	19.639	11.665	12.842	-	11.394
	Overall	-	**27.961**	**43.944**	**17.403**	**25.672**	-	**17.661**
Aspects (AOdmLib)	Size 1	0.100	0.150	0.091	0.078	0.185	0.973	0.055
	Size 2	0.165	0.767	0.736	0.121	0.446	3.875	0.132
	Size 3	0.300	5.020	7.799	0.174	3.782	5.349	0.395
	Size 4	0.866	8.813	15.634	5.350	8.511	6.578	5.734
	Size 5	3.283	13.609	19.554	11.737	12.824	9.133	11.502
	Overall	**4.714**	**28.359**	**43.814**	**17.460**	**25.748**	**25.908**	**17.818**

Table E.7: JGF - Improvements over the SM versions.

Type	Input	Crypt	Ray	Series	SOR	Sparse	MC	LUFact
Gains	Size 1	1.18	1.29	1.22	2.57	-	1.05	*
	Size 2	1.26	1.73	1.17	2.33	-	1.12	1.30
	Size 3	1.16	2.12	1.11	2.28	-	1.10	1.12
	Size 4	1.08	2.16	1.09	1.00	-	1.10	1.34
	Size 5	1.08	2.22	1.09	0.99	-	1.10	1.05
	Overall	**1.10**	**2.16**	**1.10**	**1.02**	-	**1.10**	**1.15**
Time Difference (s)	Size 1	0.013	0.065	0.024	0.088	-	0.010	0.010
	Size 2	0.033	0.540	0.121	0.110	-	0.085	0.048
	Size 3	0.031	5.564	0.845	0.151	-	0.127	0.050
	Size 4	0.044	9.866	1.457	0.015	-	0.160	1.821
	Size 5	0.163	15.883	1.704	-0.082	-	0.184	0.547
	Overall	**0.284**	**31.918**	**4.151**	**0.282**	-	**0.566**	**2.466**

- *LUFact does not scale for the Size 1;
- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table E.8: JGF - Improvements over the DM versions.

Type	Input	Crypt	Ray	Series	SOR	Sparse	MC	LUFact
Gains	Size 1	-	1.32	1.00	1.61	1.04	-	1.34
	Size 2	-	1.38	1.13	1.91	1.03	-	1.24
	Size 3	-	1.43	1.02	1.71	*	-	1.03
	Size 4	-	1.44	1.01	1.01	*	-	1.00
	Size 5	-	1.44	1.00	1.02	1.05	-	1.00
	Overall	-	**1.43**	**1.01**	**1.03**	**1.03**	-	**1.00**
Time Difference (s)	Size 1	-	0.046	0.000	0.047	0.007	-	0.015
	Size 2	-	0.288	0.093	0.110	0.015	-	0.027
	Size 3	-	2.113	0.143	0.122	*	-	0.012
	Size 4	-	3.844	0.220	0.049	*	-	-0.023
	Size 5	-	5.855	0.061	0.174	0.650	-	-0.023
	Overall	-	**12.146**	**0.517**	**0.502**	**0.672**	-	**0.008**

- *For the Size 2 and Size3 Sparse does not scale;
- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table E.9: JGF - Java *vs.* Aspects in different versions.

Version	Type	Input	Crypt	Ray	Series	SOR	Sparse	MC	LUFact
SEQ.	Gains	Size 1	1.01	1.00	1.00	1.01	0.99	1.00	0.97
		Size 2	1.00	1.00	1.00	1.01	1.00	1.00	1.00
		Size 3	1.00	1.00	1.00	1.00	1.00	1.00	1.00
		Size 4	1.00	1.00	1.00	1.01	1.00	1.00	0.95
		Size 5	1.00	1.00	1.00	1.00	1.00	1.00	0.96
		Overall	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**1.00**	**0.96**
	Time Diff. (s)	Size 1	0.001	0.000	-0.001	0.002	-0.002	0.001	-0.002
		Size 2	0.000	0.012	0.002	0.005	-0.001	-0.014	0.000
		Size 3	-0.001	-0.135	0.011	0.000	-0.012	-0.020	0.012
		Size 4	-0.009	0.411	0.051	0.199	-0.021	-0.003	-1.192
		Size 5	-0.048	-0.954	-0.022	-0.170	-0.006	-0.031	-1.816
		Overall	**-0.057**	**-0.666**	**0.041**	**0.036**	**-0.042**	**-0.067**	**-2.998**
SM	Gains	Size 1	1.08	1.04	1.15	1.04	1.07	0.99	*
		Size 2	1.06	1.03	1.01	1.07	1.11	1.04	1.06
		Size 3	1.09	1.00	1.01	1.07	1.00	1.01	1.15
		Size 4	1.06	1.01	1.01	1.01	1.00	1.01	0.99
		Size 5	1.05	1.00	1.00	1.00	1.01	1.01	1.01
		Overall	**1.06**	**1.00**	**1.00**	**1.01**	**1.01**	**1.01**	**1.00**
	Time Diff. (s)	Size 1	0.006	0.010	0.016	0.002	0.005	-0.001	*
		Size 2	0.008	0.023	0.007	0.006	0.011	0.026	0.009
		Size 3	0.017	-0.009	0.055	0.008	0.000	0.011	0.062
		Size 4	0.033	0.069	0.126	0.038	-0.004	0.016	-0.066
		Size 5	0.102	0.038	-0.011	0.045	0.015	0.025	0.072
		Overall	**0.166**	**0.131**	**0.193**	**0.099**	**0.027**	**0.077**	**0.077**
DM	Gains	Size 1	1.00	1.04	1.05	1.01	1.00	1.03	1.25
		Size 2	1.00	1.01	0.99	1.00	1.00	1.00	1.18
		Size 3	1.00	1.01	1.00	1.01	*	1.00	1.04
		Size 4	1.01	1.01	1.00	0.99	*	1.01	1.00
		Size 5	1.00	1.01	1.00	1.01	1.00	1.00	1.01
		Overall	**1.00**	**1.01**	**1.00**	**1.00**	**1.00**	**1.00**	**1.01**
	Time Diff. (s)	Size 1	0.000	0.006	0.004	0.001	0.000	0.028	0.011
		Size 2	0.000	0.007	-0.004	0.000	0.001	-0.010	0.020
		Size 3	0.001	0.062	-0.011	0.001	*	0.004	0.015
		Size 4	0.012	0.130	-0.034	-0.027	*	0.074	0.003
		Size 5	0.006	0.193	-0.085	0.082	-0.018	-0.012	0.108
		Overall	**0.019**	**0.398**	**-0.130**	**0.057**	**-0.017**	**0.084**	**0.157**

- In this case SEQ. is the Java (SEQ.) *vs.* Design rules.
- *That case study does not scale for that size;
- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table E.10: JGF - Aspects *vs.* JOMP.

Type	Input	Crypt	Ray	Series	SOR	Sparse	MC	LUFact
Gains	Size 1	1.23	1.02	1.19	1.28	1.15	1.11	*
	Size 2	1.30	1.01	1.04	1.19	1.14	1.00	0.90
	Size 3	1.22	1.01	1.00	1.14	1.07	1.01	0.75
	Size 4	1.12	1.00	1.00	1.00	1.07	1.01	1.03
	Size 5	1.07	1.00	1.00	0.99	1.02	1.01	1.02
	Overall	**1.10**	**1.00**	**1.00**	**1.00**	**1.05**	**1.01**	**1.02**
Time Diff. (s)	Size 1	0.018	0.004	0.024	0.016	0.011	0.021	*
	Size 2	0.040	0.008	0.028	0.017	0.015	-0.001	-0.017
	Size 3	0.046	0.029	-0.012	0.018	0.026	0.018	-0.116
	Size 4	0.071	-0.039	-0.035	-0.010	0.060	0.022	0.148
	Size 5	0.148	-0.054	-0.001	-0.063	0.038	0.015	0.249
	Overall	**0.323**	**-0.052**	**0.004**	**-0.022**	**0.150**	**0.075**	**0.264**

- *LUFact does not scale for the Size 1;
- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table E.11: JGF - Gains of the first touch approach and *NUMA* flag.

Version	Input	Crypt		SOR		LUFact	
		Gains	Time D.	Gains	Time D.	Gains	Time D.
C	Size 1	1.17	0.001	1.00	0.000	1.00	0.000
	Size 2	1.05	0.002	1.00	0.000	1.00	0.000
	Size 3	1.02	0.002	1.02	0.001	1.01	0.001
	Size 4	1.02	0.008	2.02	4.985	1.89	4.208
	Size 5	1.04	0.072	2.01	10.994	1.90	7.694
	Overall	**1.04**	0.085	**2.00**	15.980	**1.88**	11.903
JGF Original	Size 1	1.06	0.005	0.99	-0.002	*	*
	Size 2	0.99	-0.001	0.98	-0.003	1.00	0.000
	Size 3	1.00	-0.001	1.01	0.003	1.00	-0.001
	Size 4	1.00	-0.001	1.72	3.959	1.21	1.539
	Size 5	1.01	0.017	1.70	8.672	1.46	5.039
	Overall	**1.01**	0.019	**1.68**	12.629	**1.35**	6.577
JGF Improved	Size 1	0.96	-0.003	1.02	0.001	*	*
	Size 2	1.00	0.000	0.99	-0.001	0.99	-0.002
	Size 3	1.03	0.005	1.00	0.000	0.98	-0.009
	Size 4	1.03	0.016	1.72	3.943	1.64	3.461
	Size 5	1.01	0.018	1.67	8.381	1.54	5.576
	Overall	**1.01**	0.036	**1.68**	12.324	**1.55**	9.026
AOmpLib	Size 1	1.00	0.000	0.97	-0.002	*	*
	Size 2	1.04	0.005	0.95	-0.005	0.99	-0.002
	Size 3	0.98	-0.005	1.02	0.002	0.99	-0.004
	Size 4	0.99	-0.005	1.76	4.176	1.66	3.554
	Size 5	0.99	-0.031	1.54	6.817	1.47	4.879
	Overall	**1.00**	0.034	**1.60**	10.988	**1.51**	8.427
JOMP	Size 1	1.00	0.000	1.01	0.001	*	*
	Size 2	0.99	-0.001	0.94	-0.007	1.01	0.002
	Size 3	1.09	0.023	0.98	-0.003	0.99	-0.005
	Size 4	1.66	0.424	1.70	3.866	1.60	3.309
	Size 5	1.00	-0.003	1.69	8.585	1.50	5.284
	Overall	**1.03**	0.441	**1.68**	12.442	**1.51**	8.590

- *LUFact does not scale for the Size 1;
- For the C versions we used the first touch optimization technique;
- Gains between versions with and without flag *NUMA*/first touch;
- Time D. = Time Difference (seconds).

Table E.12: JGF - C *vs.* Java in different versions.

Version	Type	Input	Crypt	Ray	Series	SOR	Sparse	MC	LUFact
SEQ.	Gains	Size 1	1.56	-	0.37	1.24	1.04	-	2.13
		Size 2	1.26	-	0.52	1.19	1.02	-	1.45
		Size 3	1.22	-	0.61	1.09	0.98	-	1.26
		Size 4	1.21	-	0.62	1.00	0.99	-	1.25
		Size 5	1.20	-	0.63	1.01	1.00	-	1.18
		Overall	**1.21**	-	**0.62**	**1.01**	**0.99**	-	**1.21**
	Time Diff. (s)	Size 1	0.064	-	-1.661	0.040	0.009	-	0.035
		Size 2	0.197	-	-12.517	0.072	0.009	-	0.105
		Size 3	0.429	-	-101.575	0.070	-0.077	-	0.591
		Size 4	1.598	-	-198.710	-0.043	-0.042	-	4.986
		Size 5	7.044	-	-247.080	0.533	-0.060	-	6.929
		Overall	**9.332**	-	**-561.543**	**0.672**	**-0.161**	-	**12.646**
SM	Gains	Size 1	12.17	-	0.89	3.50	4.93	-	10.67*
		Size 2	3.13	-	0.62	2.44	3.00	-	5.71
		Size 3	1.95	-	0.66	1.97	1.39	-	2.65
		Size 4	1.37	-	0.67	1.11	1.22	-	1.14
		Size 5	1.19	-	0.67	1.14	1.05	-	1.21
		Overall	**1.32**	-	**0.67**	**1.14**	**1.17**	-	**1.21**
	Time Diff. (s)	Size 1	0.067	-	-0.014	0.040	0.055	-	0.058*
		Size 2	0.085	-	-0.439	0.049	0.064	-	0.132
		Size 3	0.095	-	-3.919	0.058	0.110	-	0.254
		Size 4	0.144	-	-7.739	0.552	0.146	-	0.680
		Size 5	0.321	-	-9.516	1.572	0.084	-	1.770
		Overall	**0.712**	-	**-21.627**	**2.271**	**0.459**	-	**2.894**
DM	Gains	Size 1	7.69	-	0.75	4.53	1.41	-	4.89
		Size 2	2.89	-	0.64	3.56	1.64	-	3.50
		Size 3	2.37	-	0.68	2.84	1.61*	-	2.03
		Size 4	1.83	-	0.68	1.04	1.40*	-	1.07
		Size 5	1.58	-	0.68	1.01	1.34	-	1.04
		Overall	**1.72**	-	**0.68**	**1.04**	**1.39**	-	**1.07**
	Time Diff. (s)	Size 1	0.087	-	-0.029	0.060	0.054	-	0.035
		Size 2	0.108	-	-0.411	0.087	0.174	-	0.080
		Size 3	0.173	-	-3.725	0.112	1.417*	-	0.193
		Size 4	0.388	-	-7.420	0.223	2.253*	-	0.392
		Size 5	1.209	-	-9.209	0.138	3.272	-	0.489
		Overall	**1.965**	-	**-20.794**	**0.620**	**7.170**	-	**1.189**

- *For that case study and that size the execution time used for Java was the sequential one since it was the fastest;
- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table E.13: JGF - Speedups of the SM implementations.

Version	Input	Crypt	Ray	Series	SOR	Sparse	MC	LUFact
C	Size 1	19.00	-	21.20	10.44	16.71	-	5.17
	Size 2	19.13	-	22.52	11.03	15.88	-	8.32
	Size 3	19.14	-	22.62	13.47	13.79	-	14.86
	Size 4	19.43	-	22.63	4.66	12.26	-	4.12
	Size 5	20.58	-	22.64	4.82	10.84	-	4.37
	Overall	**20.28**	**-**	**22.63**	**4.82**	**11.61**	**-**	**4.41**
JGF	Size 1	2.07	3.28	7.22	1.44	3.49	10.98	0.84
	Size 2	6.09	7.83	16.22	2.32	5.38	16.30	1.63
	Size 3	10.36	8.42	18.88	3.26	9.67	13.92	6.29
	Size 4	15.88	8.61	19.30	4.17	10.01	14.78	3.22
	Size 5	19.20	8.49	19.34	4.27	10.27	15.97	3.91
	Overall	**16.88**	**8.48**	**19.15**	**4.18**	**9.82**	**15.05**	**3.67**
JGF Improved	Size 1	2.44	4.24	8.79	3.70	-	11.57	0.97
	Size 2	7.70	13.54	18.91	5.39	-	18.28	2.11
	Size 3	12.01	17.88	20.95	7.44	-	15.31	7.06
	Size 4	17.18	18.61	21.12	4.18	-	16.27	4.30
	Size 5	20.77	18.85	21.02	4.24	-	17.57	4.12
	Overall	**18.51**	**18.34**	**20.98**	**4.25**	**-**	**16.57**	**4.22**
Aspects	Size 1	2.25	4.06	7.67	3.57	3.26	11.63	0.88
	Size 2	7.23	13.13	18.73	5.02	4.82	17.63	2.00
	Size 3	11.05	17.91	20.80	6.97	9.67	15.18	6.13
	Size 4	16.19	18.46	20.95	4.15	10.06	16.11	4.36
	Size 5	19.76	18.80	21.03	4.23	10.18	17.33	4.09
	Overall	**17.52**	**18.25**	**20.88**	**4.23**	**9.74**	**16.34**	**4.20**
JOMP	Size 1	1.84	3.99	6.45	2.80	2.84	10.46	0.88
	Size 2	5.56	13.00	18.05	4.22	4.23	17.65	2.22
	Size 3	9.08	17.80	20.83	6.10	9.07	14.97	8.14
	Size 4	14.40	18.55	20.99	4.16	9.38	15.89	4.24
	Size 5	18.46	18.87	21.04	4.25	9.96	17.19	3.99
	Overall	**15.86**	**18.29**	**20.88**	**4.23**	**9.29**	**16.13**	**4.13**

- The speedups of C versions are calculated using the lowest execution time of the C sequential code base;
- The speedups of the JGF, JGF improved, Aspects, and JOMP are calculated using the lowest execution time of the Java sequential code base with or without the design rules.

Table E.14: JGF - Speedups of the DM implementations.

Version	Input	Crypt	Ray	Series	SOR	Sparse	MC	LUFact
C	Size 1	8.77	-	22.66	9.82	1.79	-	3.44
	Size 2	13.42	-	22.87	11.03	1.87	-	7.28
	Size 3	15.19	-	22.81	13.25	1.64	-	12.24
	Size 4	16.43	-	22.80	4.44	1.39	-	3.67
	Size 5	16.67	-	22.81	4.58	1.85	-	3.44
	Overall	**16.45**	**-**	**22.81**	**4.58**	**1.67**	**-**	**3.62**
JGF	Size 1	1.78	4.97	11.11	1.67	1.26	2.31	1.08
	Size 2	5.83	9.57	16.58	1.94	1.12	3.30	2.43
	Size 3	7.83	12.51	20.32	2.98	0.94	3.65	7.35
	Size 4	10.83	12.61	20.62	4.20	0.93	3.97	4.10
	Size 5	12.65	12.74	20.86	4.49	1.31	3.52	3.75
	Overall	**11.54**	**12.54**	**20.58**	**4.33**	**1.13**	**3.58**	**3.92**
JGF Improved	Size 1	-	6.56	11.11	2.69	1.30	-	1.45
	Size 2	-	13.20	18.66	3.69	1.16	-	3.02
	Size 3	-	17.85	20.69	5.08	0.99	-	7.58
	Size 4	-	18.19	20.91	4.24	0.98	-	4.08
	Size 5	-	18.30	20.93	4.56	1.37	-	3.74
	Overall	**-**	**17.99**	**20.82**	**4.45**	**1.19**	-	**3.92**
Aspects	Size 1	1.78	6.30	10.63	2.65	1.30	2.25	1.16
	Size 2	5.83	13.08	18.76	3.69	1.16	3.31	2.56
	Size 3	7.81	17.63	20.72	5.05	1.00	3.64	7.29
	Size 4	10.68	17.93	20.95	4.26	0.97	3.93	4.08
	Size 5	12.63	18.04	21.02	4.52	1.38	3.53	3.70
	Overall	**11.50**	**17.74**	**20.88**	**4.44**	**1.18**	**3.57**	**3.89**

- The speedups of C versions are calculated using the lowest execution time of the C sequential code base;
- The speedups of the JGF, JGF improved, Aspects, and JOMP are calculated using the lowest execution time of the Java sequential code base with or without the design rules.

Table E.15: JGF - Ray : Execution times of the DM and Hybrid with 8 machines.

Input Size	Java		Aspects	
	DM	Hybrid	DM	Hybrid
1500	1.116	0.906	1.158	0.914
2000	1.792	1.367	1.838	1.396
2500	2.586	1.970	2.647	2.009
Total	5.494	4.243	5.643	4.319

Table E.16: JGF - Ray : Speedups of the DM and Hybrid with 8 machines.

Input Size	Java		Aspects	
	DM	Hybrid	DM	Hybrid
1500	79.28	97.66	76.41	96.81
2000	88.16	115.57	85.96	113.17
2500	94.96	124.65	92.77	122.23
Overall	**89.56**	**115.96**	**87.19**	**113.92**

- The speedups of Java, Aspects, and C adjust are calculated using the lowest execution time of the sequential code base with or without the design rules;
- Speedups of C are base on the execution time of its sequential version.

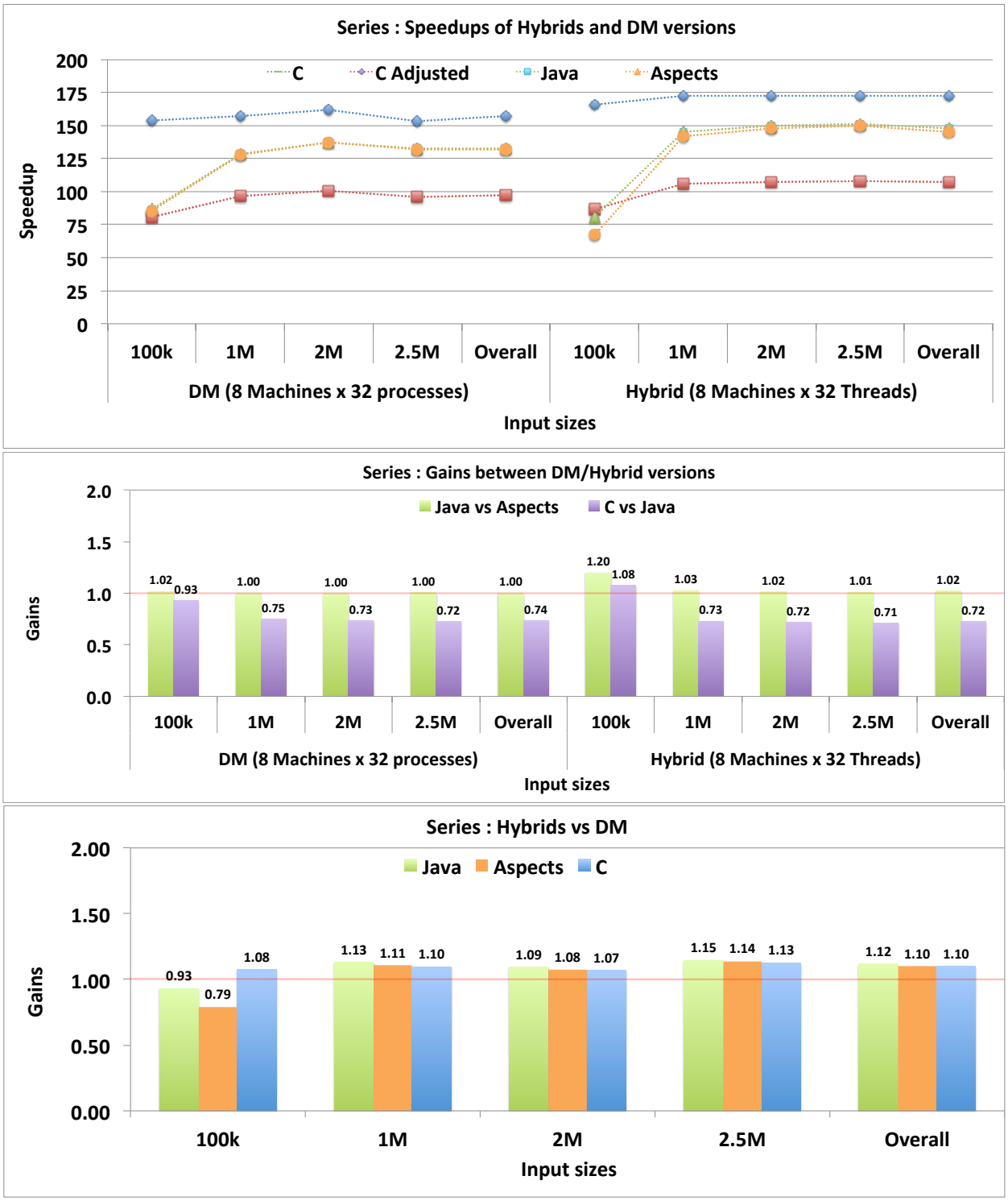


Figure E.1: JGF - Series: The tests in multi-machines, including the C implementations.

Table E.17: JGF - Series : Execution times of the DM and Hybrid with 8 machines.

Number of Particles	C		C adjust		Java		Aspects	
	DM	Hybrid	DM	Hybrid	DM	Hybrid	DM	Hybrid
100k	0.171	0.159	0.171	0.159	0.159	0.171	0.162	0.205
1M	1.676	1.528	1.676	1.528	1.258	1.113	1.261	1.141
2M	3.252	3.047	3.252	3.047	2.384	2.184	2.386	2.219
2.5M	4.293	3.809	4.293	3.809	3.107	2.713	3.120	2.742
Overall	**9.392**	**8.543**	**9.392**	**8.543**	**6.908**	**6.181**	**6.929**	**6.307**

Table E.18: JGF - Series : Speedups of the DM and Hybrid with 8 machines.

Number of Particles	C		C adjust		Java		Aspects	
	DM	Hybrid	DM	Hybrid	DM	Hybrid	DM	Hybrid
100k	153.94	165.56	80.74	86.84	86.84	80.74	85.23	67.35
1M	157.02	172.23	96.42	105.76	128.45	145.19	128.15	141.63
2M	161.84	172.73	100.74	107.51	137.41	150.00	137.30	147.63
2.5M	153.29	172.77	95.73	107.89	132.27	151.48	131.72	149.88
Overall	**156.93**	**172.52**	**97.31**	**106.98**	**132.31**	**147.87**	**131.90**	**144.91**

- The speedups of Java, Aspects, and C adjust are calculated using the lowest execution time of the sequential code base with or without the design rules;
- Speedups of C are base on the execution time of its sequential version.

Table E.19: JGF - Ray : Gains of the Hybrid *vs.* DM with 8 machines.

		Hybrid *vs.* DM	
	# of	Gains	Time Difference (s)
Java	1000	1.23	0.210
	1500	1.31	0.425
	2000	1.31	0.616
	Overall	**1.29**	1.251
Aspects	1000	1.27	0.244
	1500	1.32	0.442
	2000	1.32	0.638
	Overall	**1.31**	1.324

- Time Difference Hybrid *vs.* DM = execution time of DM - execution time of Hybrid;
- Gains Hybrid *vs.* DM = $\frac{\texttt{execution time of DM}}{\texttt{execution time of Hybrid}}$.

Table E.20: JGF - Series : Gains of the Hybrid *vs.* DM with 8 machines.

		Hybrid *vs.* DM	
	# of	Gains	Time Difference (s)
C	100k	1.08	0.012
	1M	1.10	0.148
	2M	1.07	0.205
	2.5M	1.13	0.484
	Overall	**1.10**	0.849
Java	100k	0.93	-0.012
	1M	1.13	0.145
	2M	1.09	0.200
	2.5M	1.15	0.394
	Overall	**1.12**	0.727
Aspects	100k	0.79	-0.043
	1M	1.11	0.120
	2M	1.08	0.167
	2.5M	1.14	0.378
	Overall	**1.10**	0.622

- Time Difference Hybrid *vs.* DM = execution time of DM - execution time of Hybrid;
- Gains Hybrid *vs.* DM = $\frac{\texttt{execution time of DM}}{\texttt{execution time of Hybrid}}$.

Table E.21: JGF - Ray : Comparison gains between versions for the DM and Hybrid with 8 machines.

	Input	DM		Hybrid	
	Size	Gains	Time Difference (s)	Gains	Time Difference (s)
Java *vs.* Aspects	1000	1.04	0.042	1.01	0.008
	1500	1.03	0.046	1.02	0.029
	2000	1.02	0.061	1.02	0.039
	Overall	**1.03**	0.149	**1.02**	0.076

- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Table E.22: JGF - Series : Comparison gains between versions for the DM and Hybrid with 8 machines.

	Input	DM		Hybrid	
	Size	Gains	Time Difference (s)	Gains	Time Difference (s)
Java *vs.* Aspects	100k	1.02	0.003	1.20	0.034
	1M	1.00	0.003	1.03	0.028
	2M	1.00	0.002	1.02	0.035
	2.5M	1.00	0.013	1.01	0.029
	Overall	**1.00**	0.019	**1.02**	0.126
C *vs.* Java	100k	0.93	-0.012	1.08	0.012
	1M	0.75	-0.418	0.73	-0.415
	2M	0.73	-0.868	0.72	-0.863
	2.5M	0.72	-1.186	0.71	-1.096
	Overall	**0.74**	-2.484	**0.72**	-2.362

- Time Difference A *vs.* B = execution time of B - execution time of A;
- Gains A *vs.* B = $\frac{\texttt{execution time of B}}{\texttt{execution time of A}}$.

Appendix F

JGF Benchmark : Code and Profiling results

Table F.1: JGF - RayTracer : Profiling of the sequential versions.

Version	Input	instructions	Cache references	Cache misses	Bus cycles	Branch instructions	Branch misses
Original	150	7.14E+09	1.84E+07	4.78E+06	1.67E+08	5.62E+08	1.74E+07
	500	6.83E+10	5.13E+07	1.04E+07	1.40E+09	4.31E+09	9.98E+07
	1500	5.86E+11	3.07E+08	4.44E+07	1.24E+10	3.73E+10	6.79E+08
	2000	1.07E+12	4.85E+08	7.44E+07	2.16E+10	6.69E+10	1.26E+09
	2500	1.67E+12	8.01E+08	1.32E+08	3.51E+10	1.05E+11	2.00E+09
Improved	150	4.08E+09	1.40E+07	4.48E+06	1.17E+08	3.45E+08	1.30E+07
	500	3.91E+10	4.56E+07	1.02E+07	1.04E+09	2.52E+09	8.73E+07
	1500	3.48E+11	2.85E+08	4.39E+07	9.09E+09	2.20E+10	6.56E+08
	2000	6.17E+11	5.49E+08	7.81E+07	1.64E+10	3.96E+10	1.16E+09
	2500	9.63E+11	8.67E+08	1.17E+08	2.54E+10	6.21E+10	1.78E+09
Improved *vs.* Original	150	1.75	1.31	1.07	1.43	1.63	1.34
	500	1.75	1.12	1.02	1.34	1.71	1.14
	1500	1.75	1.31	1.07	1.43	1.63	1.34
	2000	1.75	1.31	1.07	1.43	1.63	1.34
	2500	1.75	1.31	1.07	1.43	1.63	1.34

- Values measured using *perf* profiler.

Table F.2: JGF - SOR Part 1 : Profiling of the Java sequential versions.

Version	Input	Total Instructions	Branch Instructions	L1 Cache Misses
Original	1k x 1k	2.64E+09	5.23E+08	2.73E+07
	1.5k x 1.5k	5.65E+09	1.15E+09	1.13E+08
	2k x 2k	1.02E+10	2.03E+09	2.01E+08
	10k x 10k	2.45E+11	5.00E+10	5.01E+09
	15k x 15k	5.51E+11	1.12E+11	1.13E+10
Improved	1k x 1k	1.03E+09	4.84E+07	2.94E+07
	1.5k x 1.5k	2.16E+09	8.20E+07	1.15E+08
	2k x 2k	3.73E+09	1.27E+08	2.03E+08
	10k x 10k	9.02E+10	2.55E+09	5.02E+09
	15k x 15k	2.03E+11	5.68E+09	1.13E+10
Improved *vs.* Original	1k x 1k	2.57	10.79	0.93
	1.5k x 1.5k	2.62	13.97	0.98
	2k x 2k	2.74	16.05	0.99
	10k x 10k	2.72	19.64	1.00
	15k x 15k	2.72	19.78	1.00

- Values measured using *PAPI* profiler.

Table F.3: JGF - SOR Part 2 : Profiling of the Java sequential versions.

Version	Input	L3 Cache Misses	L3 Total Cache Accesses	% L3 Cache Misses	L2 Cache Misses	L2 Total Cache Accesses	% L2 Cache Misses
Original	1k x 1k	7.03E+02	1.32E+06	0.05	1.24E+06	2.73E+07	4.56
	1.5k x 1.5k	5.28E+04	5.09E+06	1.04	5.41E+06	1.13E+08	4.77
	2k x 2k	6.02E+06	2.82E+07	21.39	2.80E+07	2.01E+08	13.89
	10k x 10k	9.47E+07	1.66E+09	5.71	1.44E+09	5.01E+09	28.69
	15k x 15k	1.99E+08	4.18E+09	4.75	3.62E+09	1.13E+10	32.16
Improved	1k x 1k	9.39E+02	3.16E+06	0.03	3.41E+06	2.94E+07	11.59
	1.5k x 1.5k	9.42E+04	1.92E+07	0.49	2.05E+07	1.15E+08	17.79
	2k x 2k	8.69E+06	5.01E+07	17.35	5.17E+07	2.03E+08	25.42
	10k x 10k	2.45E+08	2.34E+09	10.46	2.28E+09	5.02E+09	45.47
	15k x 15k	7.34E+08	6.09E+09	12.06	6.00E+09	1.13E+10	53.27
Improved *vs.* Original	1k x 1k	0.75	0.42	1.79	0.37	0.93	0.39
	1.5k x 1.5k	0.56	0.26	2.11	0.26	0.98	0.27
	2k x 2k	0.69	0.56	1.23	0.54	0.99	0.55
	10k x 10k	0.39	0.71	0.55	0.63	1.00	0.63
	15k x 15k	0.27	0.69	0.39	0.60	1.00	0.60

- Values measured using *PAPI* profiler;
- % Lx Cache Misses = $\frac{\texttt{Lx Cache Misses}}{\texttt{Lx Cache Accesses}} * \texttt{100}$.

Table F.4: JGF - SOR Part 1 : Profiling of the original and improved Java SM versions.

Version	Input	Total Instructions	Branch Instructions	L1 Cache Misses	Execution time(s)
Original	1k x 1k	7.59E+09	1.11E+09	3.98E+07	0.179
	1.5k x 1.5k	1.07E+10	1.84E+09	1.19E+08	0.196
	2k x 2k	1.61E+10	2.96E+09	2.09E+08	0.285
	10k x 10k	3.03E+11	6.46E+10	5.02E+09	5.493
	15k x 15k	6.91E+11	1.55E+11	1.13E+10	12.543
Improved	1k x 1k	1.75E+09	2.44E+08	3.78E+07	0.057
	1.5k x 1.5k	4.66E+09	4.86E+08	1.17E+08	0.095
	2k x 2k	7.08E+09	9.17E+08	2.07E+08	0.131
	10k x 10k	1.59E+11	1.73E+10	5.02E+09	5.457
	15k x 15k	3.90E+11	3.82E+10	1.13E+10	12.511
Improved *vs.* Original	1k x 1k	4.33	4.56	1.05	3.14
	1.5k x 1.5k	2.30	3.79	1.01	2.06
	2k x 2k	2.27	3.23	1.01	2.18
	10k x 10k	1.91	3.73	1.00	1.01
	15k x 15k	1.77	4.05	1.00	1.00

- Values measured using *PAPI* profiler;
- All versions were tested with 12 threads and with the flag *-XX:+UseNUMA*.

Table F.5: JGF - SOR Part 2 : Profiling of the original and improved SM versions.

Version	Input	L3 Cache Misses	L3 Total Cache Accesses	% L3 Cache Misses	L2 Cache Misses	L2 Total Cache Accesses	% L2 Cache Misses
Original	1k x 1k	2.98E+06	6.77E+06	44.08	7.65E+06	3.74E+07	20.44
	1.5k x 1.5k	2.11E+06	1.44E+07	14.61	1.52E+07	1.20E+08	12.63
	2k x 2k	3.70E+06	2.18E+07	16.96	1.98E+07	2.09E+08	9.49
	10k x 10k	3.06E+08	2.19E+09	13.94	2.20E+09	5.02E+09	43.75
	15k x 15k	6.82E+08	5.27E+09	12.96	5.31E+09	1.13E+10	47.09
Improved	1k x 1k	8.19E+05	6.67E+06	12.27	7.07E+06	3.94E+07	17.93
	1.5k x 1.5k	1.13E+06	2.04E+07	5.54	2.05E+07	1.18E+08	17.43
	2k x 2k	1.56E+06	3.72E+07	4.19	3.91E+07	2.07E+08	18.91
	10k x 10k	3.69E+08	2.36E+09	15.65	2.37E+09	5.02E+09	47.11
	15k x 15k	8.55E+08	5.61E+09	15.26	5.60E+09	1.13E+10	49.64
Improved *vs.* Original	1k x 1k	3.65	1.01	3.59	1.08	0.95	1.14
	1.5k x 1.5k	1.86	0.70	2.64	0.74	1.02	0.72
	2k x 2k	2.38	0.59	4.05	0.51	1.01	0.50
	10k x 10k	0.83	0.93	0.89	0.93	1.00	0.93
	15k x 15k	0.80	0.94	0.85	0.95	1.00	0.95

- Values measured using *PAPI* profiler;
- All versions were tested with 12 threads and with the flag *-XX:+UseNUMA*;
- % Lx Cache Misses = $\frac{\texttt{Lx Cache Misses}}{\texttt{Lx Cache Accesses}} * \texttt{100}$.

Table F.6: JGF - LUFact : Profiling of the design rules gains.

Version	Input	Total Instructions	Load instructions	Store instruction	L1 Cache Misses	Execution Time (s)
Sequential	500 x 500	4.04E+08	1.65E+08	1.29E+08	5.55E+07	0.066
	1k x 1k	1.93E+09	8.78E+08	7.26E+08	3.50E+08	0.338
	2k x 2k	1.23E+10	7.46E+09	5.03E+09	2.49E+09	2.880
	4k x 4k	9.45E+10	6.90E+10	4.16E+10	1.96E+10	24.582
	16k x 16k	1.75E+11	1.37E+11	7.70E+10	3.62E+10	44.424
Design Rules	500 x 500	3.98E+08	1.62E+08	1.26E+08	5.41E+07	0.064
	1k x 1k	1.92E+09	8.78E+08	7.27E+08	3.51E+08	0.338
	2k x 2k	1.23E+10	7.49E+09	5.04E+09	2.51E+09	2.892
	4k x 4k	9.45E+10	6.14E+10	3.92E+10	1.96E+10	23.390
	16k x 16k	1.75E+11	1.11E+11	7.27E+10	3.62E+10	42.608
Sequential *vs.* Design Rules	500 x 500	0.99	0.98	0.98	0.98	0.97
	1k x 1k	1.00	1.00	1.01	1.00	1.00
	2k x 2k	1.00	1.00	1.00	1.00	1.00
	4k x 4k	1.00	0.94	1.00	0.94	0.95
	16k x 16k	1.00	0.94	1.00	0.94	0.96

- Values measured using *PAPI* profiler.

Table F.7: JGF - SOR Part 1 : Profiling of the intrusive and AOmpLib SM versions.

Version	Input	Total Instructions	Total Cycles	Branch Instructions	L1 Cache Misses	Execution time (s)
Intrusive	1k x 1k	1.75E+09	1.84E+07	1.98E+09	3.78E+07	0.057
	1.5k x 1.5k	4.66E+09	2.49E+07	3.70E+09	1.17E+08	0.095
	2k x 2k	7.08E+09	2.70E+07	5.38E+09	2.07E+08	0.131
	10k x 10k	1.59E+11	2.19E+08	1.69E+11	5.02E+09	5.457
	15k x 15k	3.90E+11	3.43E+08	3.88E+11	1.13E+10	12.511
AOmpLib	1k x 1k	1.94E+09	3.14E+07	1.84E+09	3.67E+07	0.064
	1.5k x 1.5k	3.99E+09	3.12E+07	3.31E+09	1.18E+08	0.116
	2k x 2k	6.47E+09	3.96E+07	5.16E+09	2.08E+08	0.178
	10k x 10k	1.55E+11	1.19E+08	1.70E+11	5.02E+09	5.495
	15k x 15k	3.80E+11	2.03E+08	3.89E+11	1.13E+10	12.573
Intrusive *vs.* AOmpLib	1k x 1k	1.11	1.71	0.93	0.97	1.12
	1.5k x 1.5k	0.86	1.26	0.90	1.01	1.22
	2k x 2k	0.91	1.46	0.96	1.00	1.36
	10k x 10k	0.98	0.54	1.01	1.00	1.01
	15k x 15k	0.97	0.59	1.00	1.00	1.00

- Values measured using *PAPI* profiler;
- All versions were tested with 12 threads and with the flag *-XX:+UseNUMA*.

Table F.8: JGF - SOR Part 2 : Profiling of the intrusive and AOmpLib SM versions.

Version	Input	L3 Cache Misses	L3 Total Cache Accesses	% L3 Cache Misses	L2 Cache Misses	L2 Total Cache Accesses	% L2 Cache Misses
Intrusive	1k x 1k	8.19E+05	6.67E+06	12.27	7.07E+06	3.94E+07	17.93
	1.5k x 1.5k	1.13E+06	2.04E+07	5.54	2.05E+07	1.18E+08	17.43
	2k x 2k	1.56E+06	3.72E+07	4.19	3.91E+07	2.07E+08	18.91
	10k x 10k	3.69E+08	2.36E+09	15.65	2.37E+09	5.02E+09	47.11
	15k x 15k	8.55E+08	5.61E+09	15.26	5.60E+09	1.13E+10	49.64
AOmpLib	1k x 1k	9.67E+05	8.33E+06	11.61	7.98E+06	3.72E+07	21.42
	1.5k x 1.5k	1.29E+06	2.42E+07	5.35	2.36E+07	1.19E+08	19.91
	2k x 2k	1.72E+06	3.73E+07	4.61	4.01E+07	2.08E+08	19.30
	10k x 10k	3.71E+08	2.37E+09	15.67	2.37E+09	5.02E+09	47.13
	15k x 15k	8.52E+08	5.59E+09	15.25	5.60E+09	1.13E+10	49.62
Intrusive *vs.* AOmpLib	1k x 1k	1.18	1.25	0.95	1.13	0.94	1.19
	1.5k x 1.5k	1.14	1.18	0.97	1.15	1.01	1.14
	2k x 2k	1.11	1.00	1.10	1.03	1.01	1.02
	10k x 10k	1.00	1.00	1.00	1.00	1.00	1.00
	15k x 15k	1.00	1.00	1.00	1.00	1.00	1.00

- Values measured using *PAPI* profiler;
- All versions were tested with 12 threads and with the flag *-XX:+UseNUMA*;
- % Lx Cache Misses = $\frac{\texttt{Lx Cache Misses}}{\texttt{Lx Cache Accesses}} * \texttt{100}$.

Table F.9: JGF - Crypt Part 1 : Profiling AOmpLib *vs.* JOMP.

Version	Input	Total Instructions	Branch Instructions	L1 Cache Misses	Execution Time (s)
AOmpLib	3 M	1.67E+09	1.94E+08	4.04E+06	0.10
	20 M	5.75E+09	3.08E+08	5.08E+06	0.14
	50 M	1.29E+10	5.09E+08	6.80E+06	0.21
	200 M	4.67E+10	1.23E+09	1.60E+07	0.57
	900 M	2.06E+11	4.89E+09	6.03E+07	2.10
JOMP	3 M	1.73E+09	2.06E+08	5.37E+06	0.15
	20 M	5.76E+09	3.46E+08	5.95E+06	0.18
	50 M	1.28E+10	5.54E+08	7.76E+06	0.28
	200 M	4.80E+10	1.59E+09	1.68E+07	0.64
	900 M	2.10E+11	6.41E+09	6.09E+07	2.25
AOmpLib *vs.* JOMP	3 M	1.04	1.07	1.33	1.41
	20 M	1.00	1.12	1.17	1.30
	50 M	0.99	1.09	1.14	1.33
	200 M	1.03	1.30	1.05	1.12
	900 M	1.02	1.31	1.01	1.07

- Values measured using *PAPI* profiler;
- All versions were tested with 32 threads and with the flag *-XX:+UseNUMA* for the JOMP runs.

```
1
2   ; Assembly from the "dy[i + dy_off] += da * dx[i + dx_off]
3   ; from the sequential version without the design rules
4
5   mov     %r10d,%r8d
6   add     0x48(%rsp),%r8d     ;*iinc      "The Difference!!!"
7   vmovsd  0x10(%rbx,%r8,8),%xmm0  ;*daload
8   vmulsd  %xmm1,%xmm0,%xmm0
9   vaddsd  0x10(%rdx,%r8,8),%xmm0,%xmm0
10  vmovsd  %xmm0,0x10(%rdx,%r8,8)  ;*dastore
11  movslq  %r8d,%r11
12  vmovsd  0x18(%rbx,%r11,8),%xmm0  ;*daload
13  vmulsd  %xmm1,%xmm0,%xmm0
14  vaddsd  0x18(%rdx,%r11,8),%xmm0,%xmm0
15  vmovsd  %xmm0,0x18(%rdx,%r11,8)  ;*dastore
16   ....
17  vaddsd  0x40(%rdx,%r11,8),%xmm0,%xmm0
18  vmovsd  %xmm0,0x40(%rdx,%r11,8)  ;*dastore
19  vmovsd  0x48(%rbx,%r11,8),%xmm0  ;*daload
20  vmulsd  %xmm1,%xmm0,%xmm0
21  vaddsd  0x48(%rdx,%r11,8),%xmm0,%xmm0
22  vmovsd  %xmm0,0x48(%rdx,%r11,8)  ;*dastore
23  add     $0x8,%r10d          ;*iinc
24  cmp     %edi,%r10d
25  jl      ...  ;*if_icmplt
26  ...
27
28 ;-----------------------------------------------------
29  ; Assembly from the "dy[i + dy_off] += da * dx[i + dx_off]
30  ; from the sequential version with the design rules
31
32  mov     %r13d,%eax
33  add     %r8d,%eax           ;*iadd      "The Difference!!!"
34  vmovsd  0x10(%r9,%rax,8),%xmm0  ;*daload
35  vmulsd  %xmm1,%xmm0,%xmm0
36  vaddsd  0x10(%rsi,%rax,8),%xmm0,%xmm0
37  vmovsd  %xmm0,0x10(%rsi,%rax,8)  ;*dastore
38  movslq  %eax,%rax
39  vmovsd  0x18(%r9,%rax,8),%xmm0  ;*daload
40  vmulsd  %xmm1,%xmm0,%xmm0
41  vaddsd  0x18(%rsi,%rax,8),%xmm0,%xmm0
42  vmovsd  %xmm0,0x18(%rsi,%rax,8)  ;*dastore
43  ...
44  vaddsd  0x40(%rsi,%rax,8),%xmm0,%xmm0
45  vmovsd  %xmm0,0x40(%rsi,%rax,8)  ;*dastore
46  vmovsd  0x48(%r9,%rax,8),%xmm0  ;*daload
47  vmulsd  %xmm1,%xmm0,%xmm0
48  vaddsd  0x48(%rsi,%rax,8),%xmm0,%xmm0
49  vmovsd  %xmm0,0x48(%rsi,%rax,8)  ;*dastore
50  add     $0x8,%r13d          ;*iinc
51  cmp     %r10d,%r13d
52  jl     ...  ;*if_icmplt
53  ...
```

Figure F.1: JGF - LUFact : Assembly snippet of the *daxpy* method of the sequential code with and without design rules.

```
1
2   ; Part of the assembly of "for (int i = begin; i < end; i++)" loop
3   ; from the sequential version without the for method design rule
4
5   vmulsd %xmm2,%xmm1,%xmm1
6   vaddsd %xmm1,%xmm0,%xmm0
7   movslq %eax,%rax
8   vmovsd %xmm0,0x10(%rcx,%rax,8)  ;*dastore
9   ; - design_rules.SparseMatmult::kernel@28 (line 59)
10
11  inc    %esi          ; "The Difference!!!"
12  mov    $0x2b5b71ce2780,%rax  ; ...
13  mov    0xe0(%rax),%r11d
14  add    $0x8,%r11d
15  mov    %r11d,0xe0(%rax)
16  mov    $0x2b5b71ce0240,%rax  ; ...
17  and    $0xfff8,%r11d
18  cmp    $0x0,%r11d
19  je     0x00002b5ab03a7020  ;*if_icmplt
20               ; - design_rules.SparseMatmult::kernel@35 (line 58)
21  test   %eax,-0x69e6e66(%rip)   ; ...
22  cmp    %edx,%esi
23  ...
24 ;-------------------------------------------------
25
26  ; Part of the assembly of "for (int i = begin; i < end; i+=step)" loop
27  ; from the sequential version with the for method design rule
28
29  vmulsd %xmm2,%xmm1,%xmm1
30  vaddsd %xmm1,%xmm0,%xmm0
31  movslq %r11d,%r11
32  vmovsd %xmm0,0x10(%r8,%r11,8)  ;*dastore
33  ; - design_rules.SparseMatmult::kernel@29 (line 59)
34
35  mov    %rcx,%r11          ; "The Difference!!!"
36  add    %esi,%r11d         ; "The Difference!!!"
37  mov    %r11,%rsi          ;*iload ; "The Difference!!!"
38
39  mov    $0x2ae9edce0cd0,%r11  ;  ...
40  mov    0xe0(%r11),%r13d
41  add    $0x8,%r13d
42  mov    %r13d,0xe0(%r11)
43  mov    $0x2ae9edce0100,%r11  ;  ...
44  and    $0xfff8,%r13d
45  cmp    $0x0,%r13d
46  je     0x00002ae95c3a6770  ;*if_icmplt
47               ; - design_rules.SparseMatmult::kernel@39 (line 58)
48  test   %eax,-0x5d3e5b6(%rip) ; ...
49  cmp    %edx,%esi
50  ...
```

Figure F.2: JGF - Sparse : Assembly snippet of the sparse kernel loop with and without the *for* method design rule.

Table F.10: JGF - Crypt Part 2 : Profiling AOmpLib *vs.* JOMP.

Version	Input	L3 Cache Misses	L3 Total Cache Accesses	% L3 Cache Misses	L2 Cache Misses	L2 Total Cache Accesses	% L2 Cache Misses
AOmpLib	3 M	1.18E+06	1.56E+06	75.85	3.72E+06	4.01E+06	92.643
	20 M	1.27E+06	2.06E+06	61.86	3.71E+06	5.14E+06	72.167
	50 M	1.40E+06	3.00E+06	46.52	3.89E+06	6.89E+06	56.379
	200 M	1.90E+06	7.60E+06	25.02	3.95E+06	1.59E+07	24.897
	900 M	4.82E+06	2.94E+07	16.37	7.16E+06	6.00E+07	11.918
JOMP	3 M	1.44E+06	2.17E+06	66.59	4.82E+06	5.32E+06	90.692
	20 M	1.74E+06	2.55E+06	68.06	4.59E+06	5.97E+06	76.853
	50 M	1.70E+06	3.51E+06	48.50	4.53E+06	7.72E+06	58.755
	200 M	2.12E+06	8.03E+06	26.33	5.06E+06	1.66E+07	30.437
	900 M	5.16E+06	2.98E+07	17.30	7.85E+06	6.06E+07	12.966
AOmpLib *vs.* JOMP	3 M	1.22	1.39	0.88	1.30	1.32	0.98
	20 M	1.37	1.24	1.10	1.24	1.16	1.06
	50 M	1.22	1.17	1.04	1.17	1.12	1.04
	200 M	1.11	1.06	1.05	1.28	1.05	1.22
	900 M	1.07	1.01	1.06	1.10	1.01	1.09

- Values measured using *PAPI* profiler;
- All versions were tested with 32 threads and with the flag *-XX:+UseNUMA* for the JOMP runs;
- % Lx Cache Misses = $\frac{\texttt{Lx Cache Misses}}{\texttt{Lx Cache Accesses}} * \texttt{100}$.

Table F.11: JGF - Sparse Part 1 : Profiling AOmpLib *vs.* JOMP.

Version	Input	Total Instructions	Branch Instructions	L1 Cache Misses	Execution Time (s)
AOmpLib	50k	7.48E+08	1.44E+08	9.89E+07	0.08
	100k	1.39E+09	2.68E+08	2.11E+08	0.11
	500k	6.79E+09	1.30E+09	1.09E+09	0.39
	1 M	1.34E+10	2.57E+09	2.23E+09	0.82
	1.5 M	1.99E+10	3.82E+09	3.35E+09	1.76
JOMP	50k	7.83E+08	1.47E+08	9.88E+07	0.10
	100k	1.43E+09	2.71E+08	2.11E+08	0.12
	500k	6.94E+09	1.32E+09	1.10E+09	0.45
	1 M	1.35E+10	2.58E+09	2.23E+09	0.88
	1.5 M	2.00E+10	3.84E+09	3.36E+09	1.77
AOmpLib *vs.* JOMP	50k	1.05	1.02	1.00	1.13
	100k	1.03	1.01	1.00	1.14
	500k	1.02	1.01	1.01	1.15
	1 M	1.01	1.01	1.00	1.07
	1.5 M	1.01	1.00	1.00	1.00

- Values measured using *PAPI* profiler;
- All versions were tested with fix number of threads 50k and 100k with 8 Threads and the remaining with 32.

Table F.12: JGF - Sparse Part 2 : Profiling AOmpLib *vs.* JOMP.

Version	Input	L3 Cache Misses	L3 Total Cache Accesses	% L3 Cache Misses	L2 Cache Misses	L2 Total Cache Accesses	% L2 Cache Misses
AOmpLib	50k	6.40E+05	4.50E+07	1.42	4.52E+07	9.92E+07	45.56
	100k	7.04E+05	1.30E+08	0.54	1.28E+08	2.11E+08	60.73
	500k	6.62E+07	8.56E+08	7.74	8.60E+08	1.09E+09	78.81
	1M	2.52E+08	1.99E+09	12.68	1.99E+09	2.23E+09	89.19
	1.5M	6.36E+08	3.11E+09	20.49	3.11E+09	3.35E+09	92.78
JOMP	50k	6.16E+05	4.54E+07	1.36	4.52E+07	9.88E+07	45.76
	100k	6.84E+05	1.28E+08	0.53	1.28E+08	2.12E+08	60.49
	500k	7.19E+07	8.77E+08	8.20	8.83E+08	1.10E+09	80.51
	1M	2.53E+08	2.00E+09	12.64	2.00E+09	2.23E+09	89.73
	1.5M	6.38E+08	3.12E+09	20.48	3.12E+09	3.36E+09	92.91
AOmpLib *vs.* JOMP	50k	0.96	1.01	0.95	1.00	1.00	1.00
	100k	0.97	0.99	0.98	1.00	1.01	1.00
	500k	1.09	1.02	1.06	1.03	1.01	1.02
	1M	1.00	1.00	1.00	1.01	1.00	1.01
	1.5M	1.00	1.00	1.00	1.00	1.00	1.00

- Values measured using *PAPI* profiler;
- All versions were tested with fix number of threads 50k and 100k with 8 Threads and the remaining with 32;
- % Lx Cache Misses = $\frac{\texttt{Lx Cache Misses}}{\texttt{Lx Cache Accesses}} * \texttt{100}$.

Table F.13: JGF - Crypt Part 1 : Profiling Intrusive *vs.* AOmpLib.

Version	Input	Total Instructions	Branch Instructions	L1 Cache Misses	Execution Time (s)
Intrusive	3 M	1.33E+09	1.35E+08	3.28E+06	0.095
	20 M	5.26E+09	2.50E+08	4.43E+06	0.125
	50 M	1.24E+10	4.21E+08	6.24E+06	0.195
	200 M	4.51E+10	1.09E+09	1.57E+07	0.538
	900 M	2.01E+11	4.31E+09	5.93E+07	1.996
AOmpLib	3 M	1.67E+09	1.94E+08	4.04E+06	0.104
	20 M	5.75E+09	3.08E+08	5.08E+06	0.138
	50 M	1.29E+10	5.09E+08	6.80E+06	0.212
	200 M	4.67E+10	1.23E+09	1.60E+07	0.571
	900 M	2.06E+11	4.89E+09	6.03E+07	2.098
Intrusive *vs.* AOmpLib	3 M	1.25	1.44	1.23	1.09
	20 M	1.09	1.23	1.15	1.10
	50 M	1.04	1.21	1.09	1.09
	200 M	1.03	1.12	1.02	1.06
	900 M	1.03	1.13	1.02	1.05

- Values measured using *PAPI* profiler;
- All versions were tested with 32 threads.

Table F.14: JGF - Crypt Part 2 : Profiling Intrusive *vs.* AOmpLib.

Version	Input	L3 Cache Misses	L3 Total Cache Accesses	% L3 Cache Misses	L2 Cache Misses	L2 Total Cache Accesses	% L2 Cache Misses
Intrusive	3 M	9.98E+05	2.95E+06	33.78	3.00E+06	3.24E+06	92.51
	20 M	1.06E+06	2.89E+06	36.53	2.94E+06	4.25E+06	69.03
	50 M	1.32E+06	3.25E+06	40.73	3.23E+06	6.49E+06	49.77
	200 M	1.81E+06	3.73E+06	48.51	3.79E+06	1.56E+07	24.30
	900 M	4.77E+06	6.53E+06	73.02	6.59E+06	5.94E+07	11.08
AOmpLib	3 M	1.18E+06	3.58E+06	32.98	3.72E+06	4.01E+06	92.64
	20 M	1.27E+06	3.72E+06	34.21	3.71E+06	5.14E+06	72.17
	50 M	1.40E+06	3.80E+06	36.73	3.89E+06	6.89E+06	56.38
	200 M	1.90E+06	3.82E+06	49.76	3.95E+06	1.59E+07	24.90
	900 M	4.82E+06	6.75E+06	71.37	7.16E+06	6.00E+07	11.92
Intrusive *vs.* AOmpLib	3 M	1.18	1.21	0.98	1.24	1.24	1.00
	20 M	1.20	1.29	0.94	1.26	1.21	1.05
	50 M	1.05	1.17	0.90	1.20	1.06	1.13
	200 M	1.05	1.02	1.03	1.04	1.02	1.02
	900 M	1.01	1.03	0.98	1.09	1.01	1.08

- Values measured using *PAPI* profiler;
- All versions were tested with 32 threads;
- % Lx Cache Misses = $\frac{\texttt{Lx Cache Misses}}{\texttt{Lx Cache Accesses}} * \texttt{100}$.

Table F.15: JGF - Crypt Part 1 : Profiling C *vs.* Java (sequential).

Version	Input	Total Instructions	Load instruction	Store instruction	L1 Cache Misses
C	3 M	4.70E+08	4.58E+07	6.00E+06	1.63E+05
	20 M	3.13E+09	3.05E+08	4.00E+07	1.24E+06
	50 M	7.83E+09	7.63E+08	1.00E+08	3.11E+06
	200 M	3.13E+10	3.05E+09	4.00E+08	1.25E+07
	900 M	1.41E+11	1.37E+10	1.80E+09	5.63E+07
Java	3 M	7.92E+08	1.65E+08	3.84E+07	2.21E+05
	20 M	4.70E+09	8.44E+08	1.83E+08	1.29E+06
	50 M	1.16E+10	2.04E+09	4.38E+08	3.18E+06
	200 M	4.61E+10	8.04E+09	1.71E+09	1.26E+07
	900 M	2.07E+11	3.60E+10	7.66E+09	5.66E+07
C *vs.* Java	3 M	1.69	3.60	6.41	1.36
	20 M	1.50	2.77	4.57	1.04
	50 M	1.48	2.68	4.38	1.02
	200 M	1.47	2.64	4.28	1.01
	900 M	1.47	2.63	4.26	1.01

- Values measured using *PAPI* profiler.

Table F.16: JGF - Crypt Part 2 : Profiling C *vs.* Java (sequential).

Version	Input	L3 Cache Misses	L3 Total Cache Accesses	% L3 Cache Misses	L2 Cache Misses	L2 Total Cache Accesses	% L2 Cache Misses
C	3 M	8.00E+00	1.26E+04	0.06	1.28E+04	1.72E+05	7.45
	20 M	4.43E+04	7.83E+04	56.53	7.59E+04	1.23E+06	6.11
	50 M	1.07E+05	1.87E+05	57.20	1.88E+05	3.12E+06	6.03
	200 M	4.16E+05	7.49E+05	55.55	7.48E+05	1.25E+07	5.98
	900 M	1.85E+06	3.37E+06	55.00	3.37E+06	5.63E+07	5.97
Java	3 M	1.95E+03	2.30E+04	8.47	2.40E+04	2.20E+05	10.93
	20 M	7.91E+04	8.91E+04	88.79	9.04E+04	1.29E+06	7.00
	50 M	1.90E+05	2.00E+05	94.93	2.01E+05	3.18E+06	6.33
	200 M	7.10E+05	7.80E+05	91.03	7.42E+05	1.26E+07	5.87
	900 M	3.15E+06	3.38E+06	93.15	3.29E+06	5.68E+07	5.79
C *vs.* Java	3 M	243.63	1.83	133.12	1.88	1.28	1.47
	20 M	1.79	1.14	1.57	1.19	1.05	1.15
	50 M	1.77	1.07	1.66	1.07	1.02	1.05
	200 M	1.71	1.04	1.64	0.99	1.01	0.98
	900 M	1.70	1.00	1.69	0.97	1.01	0.96

- Values measured using *PAPI* profiler;
- % Lx Cache Misses = $\frac{\texttt{Lx Cache Misses}}{\texttt{Lx Cache Accesses}} * \texttt{100}$.

Table F.17: JGF - LUFact Part 1 : Profiling C *vs.* Java (sequential).

Version	Input	Total Instructions	Load instruction	Store instruction	L1 Cache Misses
C	500^2	1.79E+08	4.41E+07	2.17E+07	5.57E+06
	$1k^2$	1.38E+09	3.43E+08	1.70E+08	4.31E+07
	$2k^2$	1.01E+10	2.51E+09	1.25E+09	3.21E+08
	$8k^2$	7.88E+10	1.96E+10	9.78E+09	4.91E+09
	$16k^2$	1.47E+11	3.63E+10	1.81E+10	9.08E+09
Java	500^2	4.04E+08	1.29E+08	5.55E+07	6.02E+06
	$1k^2$	1.93E+09	7.26E+08	3.50E+08	4.36E+07
	$2k^2$	1.23E+10	5.03E+09	2.49E+09	3.33E+08
	$8k^2$	9.45E+10	4.16E+10	1.96E+10	4.92E+09
	$16k^2$	1.75E+11	7.70E+10	3.62E+10	9.09E+09
C *vs.* Java	500^2	2.25	2.92	2.56	1.08
	$1k^2$	1.39	2.11	2.06	1.01
	$2k^2$	1.22	2.01	2.00	1.04
	$8k^2$	1.20	2.12	2.00	1.00
	$16k^2$	1.19	2.12	2.00	1.00

- Values measured using *PAPI* profiler.

Table F.18: JGF - LUFact Part 2 : Profiling C *vs.* Java (sequential).

Version	Input	L3 Cache Misses	L3 Total Cache Accesses	% L3 Cache Misses	L2 Cache Misses	L2 Total Cache Accesses	% L2 Cache Misses
C	500^2	1.00E+00	3.15E+06	0.00	3.16E+06	5.57E+06	56.66
	$1k^2$	1.70E+01	2.34E+07	0.00	2.35E+07	4.31E+07	54.56
	$2k^2$	5.53E+07	1.85E+08	29.90	1.86E+08	3.21E+08	58.13
	$8k^2$	5.95E+08	1.72E+09	34.65	1.71E+09	4.91E+09	34.82
	$16k^2$	1.38E+09	4.16E+09	33.05	4.00E+09	9.08E+09	44.11
Java	500^2	1.38E+03	2.34E+06	0.06	2.37E+06	6.11E+06	38.70
	$1k^2$	1.75E+03	1.55E+07	0.01	1.56E+07	4.37E+07	35.72
	$2k^2$	4.55E+07	1.27E+08	35.86	1.29E+08	3.34E+08	38.68
	$8k^2$	6.15E+08	1.13E+09	54.17	1.28E+09	4.92E+09	25.94
	$16k^2$	1.11E+09	3.24E-01	45.52	2.95E+09	9.09E+09	32.45
C *vs.* Java	500^2	1378.00	0.74	1854.65	0.75	1.10	0.68
	$1k^2$	103.00	0.66	155.32	0.67	1.02	0.65
	$2k^2$	0.82	0.69	1.20	0.69	1.04	0.67
	$8k^2$	1.03	0.66	1.56	0.75	1.00	0.75
	$16k^2$	0.81	0.59	1.38	0.74	1.00	0.74

- Values measured using *PAPI* profiler;
- % Lx Cache Misses = $\frac{\texttt{Lx Cache Misses}}{\texttt{Lx Cache Accesses}} * 100$.

Table F.19: JGF - Execution times of MPI Communication in C *vs.* Java.

Version	Input	Crypt	SOR	LUFact
C	Size 1	0.009	0.006	0.007
	Size 2	0.031	0.010	0.017
	Size 3	0.060	0.015	0.075
	Size 4	0.206	0.207	0.754
	Size 5	0.918	0.441	2.630
	Overall	1.224	0.679	3.483
Java	Size 1	0.016	0.017	0.031
	Size 2	0.053	0.029	0.076
	Size 3	0.102	0.043	0.204
	Size 4	0.371	0.342	0.959
	Size 5	1.718	0.678	3.075
	Overall	2.260	1.109	4.345
Gains	Size 1	1.87	2.70	4.43
	Size 2	1.69	2.82	4.47
	Size 3	1.69	2.86	2.72
	Size 4	1.80	1.65	1.27
	Size 5	1.87	1.54	1.17
	Overall	1.85	1.63	1.25
Time Difference (s)	Size 1	0.007	0.011	0.024
	Size 2	0.022	0.019	0.059
	Size 3	0.042	0.028	0.129
	Size 4	0.165	0.135	0.205
	Size 5	0.800	0.237	0.445
	Overall	1.036	0.430	0.862

```
final aspect SM_Ray extends Sm_Layer {

  declare @method: * sm.RayTracer.simulation(..): @Parallel_for_dynamic;

  declare parents : RayTracer implements PrivateObject;

  declare @method: * sm.RayTracer.simulation(..): @Private_target
  (
    data=@SmData(List = { @OP(Type = Type.SUM, vars = {"checksum"})})
  );

  public RayTracer RayTracer.copy() {return new RayTracer();}
}
```

Figure F.3: RayTracer - The best SM layer using annotations.

```
final aspect SM_SOR_user_defined_barrier extends Sm_Layer {
  ...
  static volatile long sync[][];

  // Parallel Region
  static aspect parallelRegion extends Sm_Parallel {
    ... pointcut parallel() : ( call(... void sor_simulation (..))) ...;

    @Override
    public final void before_parallel_region() {
      sync = new long[getActiveThreads()][128];
    }
  }

  // Parallel For with used defined barrier
  static aspect workSharing extends Sm_For {
    final int nthreads = getActiveThreads();

    pointcut for_static() : call(... void row_iterations(..))...;

    // Overriding the smLib barrier with an used defined one
    @Override
    public final void barrier()
    {
      final int id = getWorkerID();

      sync[id][0]++;

      if (id > 0) {
        while (sync[id-1][0] < sync[id][0]) ;
      }
      if (id < nthreads -1) {
        while (sync[id+1][0] < sync[id][0]) ;
      }
    }
  }
}
```

Figure F.4: SOR - The best SM layer using pointcuts.

```
public aspect DM extends Dm_Layer {

  pointcut hotspot() : call(... void simulation(int, int, int, ..));

  static aspect splitRow extends DM_Share<RayTracer> {

    pointcut scatter_creation() : call(... int[] row_creation(..));

    pointcut gather() : hotspot();

    pointcut get_local_index(int y) : call(... int getIndex (.., int))
                                      && args(.., y);
    @Override
    public final int chunk(RayTracer source) {
      return source.width;
    }

    ...
  }

  static aspect reduce_checksum extends Dm_Comm<RayTracer> {
    pointcut after_comm() : hotspot();

    @Override
    public final void data(RayTracer source) {
      source.checksum = reduce_to_master(source.checksum, Reduction_DM_OP.SUM);
    }
  }

  static aspect Parallel_For extends Dm_For {
    pointcut for_static() : hotspot();

    @Override
    public final int chunk() { return 1;}
  }

  static aspect filter extends Dm_Filter {
    pointcut master() : (call(... void JGFvalidate()));
  }
}
```

Figure F.5: RayTracer - The best DM layer using pointcuts.

```
public aspect DM extends Dm_Layer {

  ... static aspect splitResultsVector extends DM_Share <AppDemo>
  {
    pointcut scatter_creation() :   call(... Vector createVector());

    pointcut gather()           :   call(... void runSerial());

    @Override
    public final void data(AppDemo source) {
      source.results = create_view(source.results);
    }
  }

  static aspect Parallel_For extends Dm_For {
    pointcut for_static()  : call(... void simulation(int, int, int))
  }

  static aspect filter extends Dm_Filter {
    pointcut master(): (  call(... void JGFvalidate()) ||
                          call(... void presults ())
                       );
  }
}
```

Figure F.6: MC - The best DM layer using pointcuts.

Bibliography

[AB08] R. Arora and P. Bangalore. Using aspect-oriented programming for checkpointing a parallel application. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications, PDPTA, Las Vegas, Nevada, USA, July 14-17*, volume 2, pages 955–961, 2008.

[ABG02] D. Abramson, R. Buyya, and J. Giddy. A computational economy for grid computing and its implementation in the nimrod-g resource broker. *Future Gener. Comput. Syst.*, 18(8):1061–1074, October 2002.

[ABV92] M. Aksit, L. Bergmans, and S. Vural. An object-oriented language-database integration model: The composition-filters approach. In *Proceedings of the European Conference on Object-Oriented Programming*, ECOOP, pages 372–395, London, 1992. Springer-Verlag.

[ABVM10] D. Ansaloni, W. Binder, A. Villazón, and P. Moret. Parallel dynamic analysis on multicores with aspect-oriented programming. In *Proceedings of the 9th International Conference on Aspect-Oriented Software Development*, AOSD, pages 1–12, NY, USA, 2010. ACM.

[ACN09] S. Akai, S. Chiba, and M. Nishizawa. Region pointcut for aspectj. In *Proceedings of the 8th Workshop on Aspects, Components, and Patterns for Infrastructure Software*, ACP4IS '09, pages 43–48, New York, NY, USA, 2009. ACM.

[ADT03] M. Aldinucci, M. Danelutto, and P. Teti. An advanced environment supporting structured parallel programming in java. *Future Gener. Comput. Syst.*, 19(5):611–626, 2003.

[AE06] M. Aho and A. Eaddy. Statement annotations for fine-grained advising. In *RAM-SE'06–ECOOP'06 Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, page 89, 2006.

[AESC08] V. Arnaoudova, L. M. Eshkevar, E. S. Sharifabadi, and C. Constantinides. Overcoming comprehension barriers in the aspectj programming language. *Journal of Object Technology*, 7(6):121–142, July 2008.

[AHO+07] Pavel Avgustinov, Elnar Hajiyev, Neil Ongkingco, Oege de Moor, Damien Sereni, Julian Tibble, and Mathieu Verbaere. Semantics of static pointcuts in aspectj. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 11–23, NY, USA, 2007. ACM.

[Ale00] Milenkovic Aleksandar. Achieving high performance in bus-based shared memory multiprocessors. *IEEE CONCURRENCY*, 8(3):36–44, 2000.

[ALT08] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342 – 5359, 2008.

[Ape07] Sven Apel. *The role of features and aspects in software development: similarities, differences, and synergetic potential*. PhD thesis, Otto-von-Guericke University Magdeburg, Germany, 2007.

[apiva] OpenMP application programming interface : version 3.1. `www.openmp.org/wp-content/uploads/OpenMP3.1.pdf`. Accessed: 2017-06-16.

[apivb] OpenMP application programming interface : version 4.0. `http://www.openmp.org/wp-content/uploads/OpenMP4.0.0.pdf`. Accessed: 2017-06-16.

[Asp] AspectGrid. `https://alba.di.uminho.pt/research/projects/aspectgrid/`. Accessed: 2018-06-16.

[AtJF] AspectJ and AspectWerkz to Join Forces. `http://www.eclipse.org/aspectj/aj5announce.html`.

[B.04] Ron B. Enterprise security aspects. In *Proceedings of the 2008 ACM Symposium on Applied Computing*. AOSD, 2004.

[Ban07] Purushotham V Bangalore. Generating parallel applications for distributed memory systems using aspects, components, and patterns. In *Proceedings of the 6th workshop on Aspects, components, and patterns for infrastructure software*, page 3. ACM, 2007.

[BCF+99] Mark Baker, Bryan Carpenter, Geoffrey Fox, Sung Hoon Ko, and Sang Lim. mpijava: An object-oriented java interface to mpi. In *International Parallel Processing Symposium*, pages 748–762. Springer, 1999.

[BDGS92] J. K. Bennett, S. Dwarkadas, J. Greenwood, and E. Speight. Willow: a scalable shared memory multiprocessor. In *Proceedings of the 1992 ACM/IEEE conference on Supercomputing*, Supercomputing '92, pages 336–345, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[Ber94] L. Bergmans. *Composing Concurrent Objects - Applying Composition Filters for the Development and Reuse of Concurrent Object-Oriented Programs*. PhD thesis, University of Twente, Netherlands, 1994.

[BFN13] Dick Buttlar, Jacqueline Farrell, and Bradford Nichols. *PThreads Programming : A POSIX Standard for Better Multiprocessing*. O'Reilly Media, 2013.

[BH02] Jason Baker and Wilson Hsieh. Runtime aspect weaving through metaprogramming. In *Proceedings of the 1st International Conference on Aspect-oriented Software Development*, AOSD '02, pages 86–95, NY, USA, 2002. ACM.

[BH08] Marc Bartsch and Rachel Harrison. An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Control*, 16(1):23–44, March 2008.

[BHMO04] C. Bockisch, M. Haupt, Mira Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD : Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, NY, USA, 2004. ACM.

[BK00] J. M. Bull and M. E. Kambites. Jomp - an openmp-like interface for java. In *Proceedings of the ACM Conference on Java Grande*, JAVA, pages 44–53, NY, USA, 2000. ACM.

[BLJT07] M. Bynens, B. Lagaisse, W. Joosen, and E. Truyen. The elementary pointcut pattern. In *Proceedings of the 2Nd Workshop on Best Practices in Applying Aspect-oriented Software Development*, BPAOSD, NY, USA, 2007. ACM.

[BM08] H. Brunst and B. Mohr. Performance analysis of large-scale openmp and hybrid mpi/openmp applications with vampir ng. In M. S. Mueller, B. M. Chapman, B. R. de Supinski, A. D. Malony, and M. Voss, editors, *OpenMP Shared Memory Parallel Programming*, pages 5–14. Springer Berlin Heidelberg, 2008.

[BME07] A. Basumallik, S. Min, and R. Eigenmann. Programming distributed memory sytems using openmp. In *Parallel and Distributed Processing Symposium, IPDPS*, pages 1–8. IEEE, 2007.

[Bon04] J. Boner. Aspectwerkz - dynamic aop for java. In *Proceeding of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster, UK, 2004.

[BPP+02] L S. Blackford, A. Petitet, R. Pozo, K. Remington, R C. Whaley, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, et al. An updated set of basic linear algebra subprograms (blas). *ACM Transactions on Mathematical Software*, 28(2):135–151, 2002.

[Bre09] Clay Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc., 2009.

[BS03] László Böszörményi and Peter Schojer, editors. *Modular Programming Languages, Joint Modular Languages Conference, JMLC 2003, Klagenfurt, Austria, August 25-27, 2003, Proceedings*, volume 2789 of *Lecture Notes in Computer Science*. Springer, 2003.

[BSR04] D. Batory, J. N. Sarvela, and A. Rauschmayer. Scaling step-wise refinement. *IEEE Trans. Softw. Eng.*, 30(6):355–371, June 2004.

[BTF04] Ohad Barzilay, Shmuel Tyszberowicz, and Yishai A. Feldman. Call and execution semantics in aspectj. In G. T. Leavens, C. Clifton, , and R. Lämmel, editors, *FOAL 2004 Proceedings - Foundations of Aspect-Oriented Languages - Workshop at AOSD 2004.*, Department of Computer Science, Iowa State University, March 2004.

[Bug] Aspectj Bug. `https://bugs.eclipse.org/bugs/show_bug.cgi?id=318878`. Accessed: 2018-06-16.

[Byn11] M. Bynens. *A System of Patterns for the Design of Reusable Aspect Libraries*. PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, October 2011. Joosen, Wouter (supervisor), Truyen, Eddy (cosupervisor).

[BZS93] Brian N. Bershad, Matthew J. Zekauskas, and Wayne A. Sawdon. The midway distributed shared memory system. Technical report, Carnegie Mellon University, Pittsburgh, PA, USA, 1993.

[CA08] K. Cwalina and B. Abrams. *Framework Design Guidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries*. Addison-Wesley Professional, 2nd edition, 2008.

[Car] OpenMP 4.0 API C/C++ Syntax Quick Reference Card. `http://www.openmp.org/wp-content/uploads/OpenMP-4.0-C.pdf`. Accessed: 2017-06-16.

[cat] Refactoring catalog. `http://refactoring.com/catalog/extractMethod.html`. Accessed: 2018-06-16.

[CC07] Kung Chen and C.-H Chien. Extending the field access pointcuts of aspectj to arrays. *Journal of Software Engineering Studies*, 2:93–102, 01 2007.

[CCHW04] A. Colyer, A. Clement, G. Harley, and M. Webster. *Eclipse Aspectj: Aspect-oriented Programming with Aspectj and the Eclipse Aspectj Development Tools*. Addison-Wesley Professional, first edition, 2004.

[CCS95] Kenneth Cameron, Lyndon J Clarke, and A Gordon Smith. Cri/epcc mpi for cray t3d. In *1st European Cray T3D Workshop, EPFL*, 1995.

[CCvEK97] Chi-Chao Chang, Grzegorz Czajkowski, Thorsten von Eicken, and Carl Kesselman. Evaluating the performance limitations of mpmd communication. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '97, pages 1–10, NY, USA, 1997. ACM.

[CCZ07] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, August 2007.

[CD01] Yiannis Cotronis and Jack Dongarra, editors. *The SPMD Model: Past, Present and Future*, pages 1–1. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[CEtPG11] NVIDIA CAPS Enterprise, Cray Inc. and the Portland Group. The openacc application programming interface, v1.0, November 2011.

[CG02] S. Canditt and M. Gunter. Aspect oriented logging in a real-world system. In *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-2002)*, March 2002.

[CGS+05] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: An object-oriented approach to non-uniform cluster computing. *SIGPLAN Not.*, 40(10):519–538, October 2005.

[CGS+09] R. Chitchyan, P. Greenwood, A. Sampaio, A. Rashid, A. Garcia, and L. F. da Silva. Semantic vs. syntactic compositions in aspect-oriented requirements engineering: An empirical study. In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD, pages 149–160, NY, USA, 2009. ACM.

[CI05] Shigeru Chiba and Rei Ishikawa. Aspect-oriented programming beyond dependency injection. In Andrew P. Black, editor, *ECOOP 2005 - Object-Oriented Programming*, pages 121–143, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[CJ03] María Agustina Cibrán and Viviane Jonckers. Aspect-oriented programming for connecting business rules. In *Proceedings of the 6th International Conference on Business Information Systems, Colorado*, 2003.

[CKPD99] Henri Casanova, Myungho Kim, James S. Plank, and Jack J. Dongarra. Adaptive scheduling for task farming with grid middleware. *Int. J. High Perform. Comput. Appl.*, 13(3):231–240, August 1999.

[CL15] R. Clucas and S. Levitt. Capp: A c++ aspect-oriented based framework for parallel programming with opencl. In *Proceedings of the Annual Research Conference on South African Institute of Computer Scientists and Information Technologists*, SAICSIT, pages 10:1–10:10, NY, USA, 2015. ACM.

[Cla] Java Concurrent Atomic Classes. `https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html`. Accessed: 2018-06-16.

[Cli05] C. Clifton. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical report, Department of Computer Science, Iowa State University, 226 Atanasoff Hall, Ames, Iowa 50011, July 2005.

[CM08] Gilberto Contreras and Margaret Martonosi. Characterizing and improving the performance of intel threading building blocks. *2008 IEEE International Symposium on Workload Characterization*, pages 57–66, 2008.

[CMT04] S. Cahon, N. Melab, and E.-G. Talbi. Paradiseo: A framework for the reusable design of parallel and distributed metaheuristics. *Journal of Heuristics*, 10(3):357–380, May 2004.

[Col91] Murray Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, Cambridge, MA, USA, 1991.

[CRB04] Adrian Colyer, Awais Rashid, and Gordon Blair. On the separation of concerns in program families. Technical report, Lancaster University, 2004.

[CS07] C. A Cunha and J. Luís Sobral. An annotation-based framework for parallel computing. In *Parallel, Distributed and Network-Based Processing, 2007. PDP'07. 15th EUROMICRO International Conference on*, pages 113–120. IEEE, 2007.

[CSM06] Carlos A. Cunha, João L. Sobral, and Miguel P. Monteiro. Reusable aspect-oriented implementations of concurrency patterns and mechanisms. In *Proceedings of the 5th international conference on Aspect-oriented software development*, AOSD '06, pages 134–145, NY, USA, 2006. ACM.

[CT04] Mariano Ceccato and Paolo Tonella. Adding distribution to existing applications by means of aspect oriented programming. In *Source Code Analysis and Manipulation, 2004. Fourth IEEE International Workshop on*, pages 107–116. IEEE, 2004.

[CV13] W. Cazzola and E. Vacchi. Fine-grained annotations for pointcuts with a finer granularity. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, pages 1706–1711. ACM, 2013.

[DHP08] Vassilios V. Dimakopoulos, Panagiotis E. Hadjidoukas, and Giorgos Ch. Philos. A microbenchmark study of openmp overheads under nested parallelism. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 1–12, Berlin, Heidelberg, 2008. Springer-Verlag.

[Dij79] Edsger Dijkstra. Programming considered as a human activity. In *Classics in software engineering*, pages 1–9. Yourdon Press, 1979.

[DM98] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *IEEE Comput. Sci. Eng.*, 5(1):46–55, January 1998.

[DM14] John S. Dean and Frank J. Mitropoulos. An aspect pointcut for parallelizable loops. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, SAC '14, pages 1619–1624, NY, USA, 2014. ACM.

[DMCN12] Javier Diaz, Camelia Munoz-Caro, and Alfonso Nino. A survey of parallel programming models and tools in the multi and many-core era. *IEEE Trans. Parallel Distrib. Syst.*, 23(8):1369–1386, August 2012.

[DPV+08] M. Danelutto, M. Pasin, M. Vanneschi, P. Dazzi, D. Laforenza, and L. Presti. *PAL: Exploiting Java Annotations for Parallelism*, pages 83–96. Springer US, Boston, MA, 2008.

[DRM] J. Dongarra, Wade R., and P. McMahan. Linpack. `http://www.netlib.org/benchmark/linpackjava/`. Accessed: 28-03-2018.

[DSK05] T Fountain D Sima and P Kacsuk. *Advanced Computer Architectures : a design space approach*. Addison-Wesley, 2005.

[DWWW05] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. Polyaml: A polymorphic aspect-oriented functional programming language. *SIGPLAN Not.*, 40(9):306–319, September 2005.

[ea09] D. Binkley et al. Tool-supported refactoring of existing object-oriented code into aspects. *IEEE Transactions on Software Engineering*, 32(9):698–717, 2009.

[EBFJ16] Pacôme Eberhart, Julien Brajard, Pierre Fortin, and Fabienne Jézéquel. *Estimation of Round-off Errors in OpenMP Codes*, pages 3–16. Springer International Publishing, Cham, 2016.

[Ecl18a] Eclipse. Frequently asked questions. `https://www.eclipse.org/aspectj/doc/released/faq.php`, 2018. Accessed: 01-05-2018.

[Ecl18b] Eclipse. Inter-type declarations and annotations. `https://www.eclipse.org/aspectj/doc/next/adk15notebook/annotations-itds.html`, 2018. Accessed: 01-05-2018.

[EGCSY03] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick. *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[EMR09] P. Evangelista, P. Maia, and M. Rocha. Implementing metaheuristic optimization algorithms with jecoli. In *Proceedings of the Ninth International Conference on Intelligent Systems Design and Applications*, ISDA, pages 505–510, DC, USA, 2009. IEEE.

[ERAEB05] H. El-Rewini and M. Abd-El-Barr. *Advanced Computer Architecture and Parallel Processing (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2005.

[Fab05] J Fabry. *Modularizing Advanced Transaction Management - Tackling Tangled Aspect Code*. PhD thesis, Vrije Universiteit Brussel, Belgium, 2005.

[FBB+99] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[FDNT15] J. Fabry, T. Dinkelaker, J. Noyé, and É. Tanter. A taxonomy of domain-specific aspect languages. *ACM Comput. Surv.*, 47(3):40:1–40:44, February 2015.

[FF00] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. Technical report, RIACS, 2000.

[Fla] NUMA Flag. `https://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html`. Accessed: 2018-06-16.

[For94] Message P Forum. Mpi: A message-passing interface standard. Technical report, University of Tennessee, Knoxville, TN, USA, 1994.

[For12] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard Version 3.0, Sep. 2012. Chapter author for Collective Communication, Process Topologies, and One Sided Communications.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[FPS15] Naznin Fauzia, Louis-Noël Pouchet, and P. Sadayappan. Characterizing and enhancing global memory data coalescing on gpus. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 12–22, Washington, DC, USA, 2015. IEEE Computer Society.

[FSP06] J. F. Ferreira, J. L. Sobral, and A. J. Proença. Jaskel: A java skeleton-based framework for structured cluster and grid computing. In *Cluster Computing and the Grid, CCGRID. Sixth IEEE International Symposium on*, volume 1, 2006.

[GFB+04] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, September 2004.

[GHD00] W. L. George, J. G. Hagedorn, and J. E. Devaney. Impi: Making mpi interoperable. In *Journal of research of the National Institute of Standards and Technology*, 2000.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[GHTL14] W. Gropp, T. Hoefler, R. Thakur, and E. Lusk. *Using Advanced MPI: Modern Features of the Message-Passing Interface*. The MIT Press, 2014.

[GK01] Stephan Gudmundson and Gregor Kiczales. Addressing practical software development issues in aspectj with a pointcut interface. In *In Advanced Separation of Concerns*, 2001.

[GL03] J. D. Gradecki and N. Lesiecki. *Mastering AspectJ: Aspect-Oriented Programming in Java*. John Wiley & Sons, Inc., NY, USA, 2003.

[GLS14] W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014.

[GS00] L. Grippo and M. Sciandrone. On the convergence of the block nonlinear gauss-seidel method under convex constraints. *Oper. Res. Lett.*, 26(3):127–136, April 2000.

[GS09] R. Gonçalves and J. Sobral. Pluggable parallelization. *High Performance Distributed Computing, (HPDC'09), Munique, ACM Press*, June 2009.

[GS12] R. C. Gonçalves and J. L. Sobral. Modular and non-invasive distributed memory parallelization. In *Proceedings of the workshop on Modularity in Systems Software*, pages 33–38. ACM, 2012.

[GSBW09] GACP Ganegoda, D Samaranayake, L Bandara, and KADNK Wimalawarne. Jconqurr-a multicore programming toolkit for java. *International Journal of Computer and Information Engineering*, 3(4):223–230, 2009.

[GTM96] A. Grujíc, M. Tomasevíc, and V. Milutinovíc. A simulation study of hardware-oriented dsm approaches. *IEEE Parallel Distrib. Technol.*, 4(1):74–83, March 1996.

[GVL10] H. G. Vélez and M. Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40(12):1135–1160, November 2010.

[Har06] B. Harbulot. *Separating concerns in scientific software using aspect-oriented programming*. PhD thesis, University of Manchestery, 2006.

[HC02] S. Hanenberg and P. Costanza. Connecting aspects in AspectJ: Strategies vs. patterns. In *First AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software*, March 2002.

[HDB+13] T. Hoefler, J. Dinan, Darius Buntinas, Pavan Balaji, B. Barrett, R. Brightwell, W. D. Gropp, V. Kale, and Rajeev Thakur. Mpi + mpi: A new hybrid approach to parallel programming with mpi plus shared memory. *Computing*, 95:1121–1136, 2013.

[HDT+15] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. Remote memory access programming in mpi-3. *ACM Transactions on Parallel Computing*, 2(2):9, 2015.

[HE08] Kevin Hoffman and Patrick Eugster. Towards reusable components with aspects: An empirical study on modularity and obliviousness. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 91–100, NY, USA, 2008. ACM.

[HFM88] Debra Hensgen, Raphael Finkel, and Udi Manber. Two algorithms for barrier synchronization. *Int. J. Parallel Program.*, 17(1):1–17, February 1988.

[HG06] Bruno Harbulot and John R. Gurd. A join point for loops in aspectj. In *Proceedings of the 5th International Conference on Aspect-oriented Software Development*, AOSD '06, pages 63–74, NY, USA, 2006. ACM.

[HH04] Erik Hilsdale and Jim Hugunin. Advice weaving in aspectj. In *Proceedings of the 3rd International Conference on Aspect-oriented Software Development*, AOSD '04, pages 26–35, NY, USA, 2004. ACM.

[HJ09] Uwe D.C. Hohenstein and Michael C. Jäger. Using aspect-orientation in industrial projects: Appreciated or damned? In *Proceedings of the 8th ACM International Conference on Aspect-oriented Software Development*, AOSD '09, pages 213–222, NY, USA, 2009. ACM.

[HNP98] David Holmes, James Noble, and John Potter. Toward reusable synchronisation for object-oriented languages. In *Object-Oriented Technology, ECOOP'98 Workshop Reader, Demos, and Posters, Brussels, Belgium, July 20-24*, page 439, 1998.

[HO93] William Harrison and Harold Ossher. Subject-oriented programming: A critique of pure objects. *SIGPLAN Not.*, 28(10):411–428, October 1993.

[HOF+12] Ruud Haring, Martin Ohmacht, Thomas Fox, Michael Gschwind, David Satterfield, Krishnan Sugavanam, Paul Coteus, Philip Heidelberger, Matthias Blumrich, Robert Wisniewski, alan gara, George Chiu, Peter Boyle, Norman Chist, and Changhoan Kim. The ibm blue gene/q compute chip. *IEEE Micro*, 32(2):48–60, March 2012.

[HS03] Stefan Hanenberg and Arno Schmidmeier. Aspectj idioms for aspect-oriented software construction. In Kevlin Henney and Dietmar Schütz, editors, *EuroPLoP*, pages 617–644. UVK - Universitaetsverlag Konstanz, 2003.

[HT00] Weicheng Huang and Danesh Tafti. *A Parallel computing framework for dynamic power balancing in adaptive mesh refinement applications*. North Holland, 12 2000.

[HU01] S. Hanenberg and R. Unland. Using and reusing aspects in aspectj. In *In OOPSLA Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, 2001.

[IJ11] Yasuhiro Idomura and Sébastien Jolliet. Performance evaluations of advanced massively parallel platforms based on gyrokinetic toroidal five-dimensional eulerian code gt5d. *Progress in Nuclear Science and Technology*, 2:620–627, 10 2011.

[JBo] JBossAOP. `http://jbossaop.jboss.org/`. Accessed: 2018-06-16.

[JJM+11] Haoqiang Jin, Dennis Jespersen, Piyush Mehrotra, Rupak Biswas, Lei Huang, and Barbara Chapman. High performance computing using mpi and openmp on multi-core parallel systems. *Parallel Comput.*, 37(9):562–575, September 2011.

[JMOA08] Ulises Juárez-Martínez and José Oscar Olmedo-Aguirre. Énfasis: A model for local variable crosscutting. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 261–265, NY, USA, 2008. ACM.

[Jon24] J. E. Jones. On the Determination of Molecular Fields. II. From the Equation of State of a Gas. *Proceedings of the Royal Society of London Series A*, 106:463–477, 1924.

[KAB07] Christian Kastner, Sven Apel, and Don Batory. A case study implementing features using aspectj. In *Proceedings of the 11th International Software Product Line Conference*, SPLC '07, pages 223–232, Washington, DC, USA, 2007. IEEE Computer Society.

[Kam07] Alan Kaminsky. Parallel java: A unified api for shared memory and cluster parallel programming in 100 In *In 21st IEEE International Parallel and Distributed Processing Symposium (IPDPS*, 2007.

[KBVP07] Michael Klemm, Matthias Bezold, Ronald Veldema, and Michael Philippsen. Jamp: An implementation of openmp for a java dsm: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(18):2333–2352, December 2007.

[KCDZ94] Pete Keleher, Alan L. Cox, Sandhya Dwarkadas, and Willy Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *In proceeding of the 1994 winter usenix conference*, pages 115–131, 1994.

[KG02] Jörg Kienzle and Rachid Guerraoui. Aop: Does it make sense? the case of concurrency and failures. In *Proceedings of the 16th European Conference on Object-Oriented Programming*, ECOOP '02, pages 37–61, London, UK, UK, 2002. Springer-Verlag.

[KHH+01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of aspectj. In *Proceedings of the 15th European Conference on Object-Oriented Programming*, ECOOP '01, pages 327–353, London, UK, UK, 2001. Springer-Verlag.

[Kic96] G. Kiczales. Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es), 1996.

[Kis02] I. Kiselev. *Aspect-Oriented Programming with AspectJ.* Sams, IN, USA, 2002.

[KM99] Mik A. Kersten and Gail C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. Technical report, University of British Columbia, Vancouver, BC, Canada, Canada, 1999.

[Kru92] Charles W. Krueger. Software reuse. *ACM Comput. Surv.*, 24(2):131–183, June 1992.

[KS10] F. Berg Kjolstad and M. Snir. Ghost cell pattern. In *Proceedings of the 2010 Workshop on Parallel Programming Patterns*, ParaPLoP, pages 4:1–4:9, NY, USA, 2010. ACM.

[KSG09] Philipp Kegel, Maraike Schellmann, and Sergei Gorlatch. Using openmp vs. threading building blocks for medical imaging on multi-cores. In *Proceedings of the 15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 654–665, Berlin, Heidelberg, 2009. Springer-Verlag.

[Kuc04] Partha Kuchana. *Software architecture design patterns in Java.* CRC Press, 2004.

[KVBP08] M. Klemm, R. Veldema, M. Bezold, and M. Philippsen. A proposal for openmp for java. In *OpenMP Shared Memory Parallel Programming*, pages 409–421. Springer Berlin Heidelberg, 2008.

[Lad03] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming.* Manning Publications Co., Greenwich, CT, USA, 2003.

[Lad09] R. Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications.* Manning Publications Co., Greenwich, CT, USA, 2nd edition, 2009.

[Lam13] Christoph Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40:40–40:51, July 2013.

[Law98] Ramon Lawrence. A survey of cache coherence mechanisms in shared memory multiprocessors, 1998.

[LHBL06] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 68–77, New York, NY, USA, 2006. ACM.

[LJ06] Bert Lagaisse and Wouter Joosen. Decomposition into elementary pointcuts: A design principle for improved aspect reusability. In *In SPLAT*, 2006.

[LMM91] Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In Donald W. Davies, editor, *Advances in Cryptology — EUROCRYPT '91*, pages 17–38, Berlin, Heidelberg, 1991. Springer Berlin Heidelberg.

[Lop97] Cristina Lopes. *A Language Framework for Distributed Programming.* PhD thesis, College of Computer Science, Northeastern University, Boston, USA, 1997.

[LR88] K. J. Lieberherr and A. J. Riel. Demeter: A case study of software growth through parameterized classes. In *Proceedings of the 10th International Conference on Software Engineering*, ICSE '88, pages 254–264, Los Alamitos, CA, USA, 1988. IEEE Computer Society Press.

[LSB88] Thomas J. LeBlanc, Michael L. Scott, and Christopher M. Brown. Large-scale parallel programming: Experience with bbn butterfly parallel processor. *SIGPLAN Not.*, 23(9):161–172, January 1988.

[LSM11] Y. Liu, B. Schmidt, and D. L. Maskell. An ultrafast scalable many-core motif discovery algorithm for multiple gpus. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 428–434, May 2011.

[LW17] Pierre Lavallee and Philippe Wautelet. Hybrid mpi-openmp programming. `http://www.idris.fr/media/eng/formations/hybride/hybride_v3-0_en.pdf`, 2017. Accessed: 22-04-2018.

[Mar03] Robert Cecil Martin. *Agile Software Development: Principles, Patterns, and Practices.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003.

[MCS00] Mathematics and Argonne National Laboratory Computer Science. Mpi context. `https://www.mcs.anl.gov/research/projects/mpi/mpi-standard/mpi-report-1.1/node93.htm`, 2000. Accessed: 08-04-2018.

[MD09] Adrian Moga and Michel Dubois. A comparative evaluation of hybrid distributed shared-memory systems. *J. Syst. Archit.*, 55(1):43–52, January 2009.

[MEB99] Luis Moura, Silva E, and Rajkumar Buyya. Parallel programming models and paradigms. *In Rajkummar Buyya, editor, High Performance Cluster Computing: , volume 2, Programming and Applications.*, pages 4–27, 1999.

[Meu97] Wolfgang De Meuter. Monads as a theoretical foundation for aop. In *International Workshop on Aspect-Oriented Programming at ECOOP*, page 25. Springer-Verlag, 1997.

[MF04] M. Monteiro and J. Fernandes. Pitfalls of aspectj implementations of some of the gang-of-four design patterns. In *Proceedings of the workshop at JISBD 2004 (IX Jornadas de Ingeniería de Software y Bases de Datos), Málaga, Spain,*, ISDA '09, November 2004.

[MF05] M. P. Monteiro and J. M. Fernandes. Towards a catalog of aspect-oriented refactorings. In *Proceedings of the 4th International Conference on Aspect-oriented Software Development*, AOSD '05, pages 111–122, NY, USA, 2005. ACM.

[Mil04] R. Miles. *AspectJ Cookbook*. O'Reilly Media, Inc., 2004.

[Mit14] Sparsh Mittal. A study of successive over-relaxation method parallelisation over modern hpc languages. *Int. J. High Perform. Comput. Netw.*, 7(4):292–298, June 2014.

[MK03] H. Masuhara and K. Kawauchi. Dataflow pointcut in aspect-oriented programming. In A. Ohori, editor, *Programming Languages and Systems*, pages 105–121. Springer Berlin Heidelberg, 2003.

[Ml07] L. Madeyski and L. Sza la. Impact of aspect-oriented programming on software development efficiency and design quality: an empirical study. *IET Software*, 1(5):180–187, 2007.

[MLAV10] Vladimir Marjanović, Jesús Labarta, Eduard Ayguadé, and Mateo Valero. Overlapping communication and computation by using a hybrid mpi/smpss approach. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, pages 5–16, NY, USA, 2010. ACM.

[MLWR01] Gail C. Murphy, Albert Lai, Robert J. Walker, and Martin P. Robillard. Separating features in source code: An exploratory study. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 275–284, Washington, DC, USA, 2001. IEEE Computer Society.

[MMPS09] Michele Mazzucco, Graham Morgan, Fabio Panzieri, and Craig Sharp. Engineering distributed shared memory middleware for java. In *Proceedings of the Confederated International Conferences, CoopIS, DOA, IS, and ODBASE 2009 on On the Move to Meaningful Internet Systems: Part I*, OTM '09, pages 531–548, Berlin, Heidelberg, 2009. Springer-Verlag.

[Mon04] M. P. Monteiro. Catalogue of refactorings for aspectj. *Universidade do Minho, Tech. Rep. UM-DI-GECSD-200402*, 2004.

[MRR12] Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012.

[MS11] B. Medeiros and J. L. Sobral. Checkpoint and run-time adaptation with pluggable parallelisation. In *ICPP'11*, pages 434–443, 2011.

[MS13a] B. Medeiros and J. L. Sobral. Aomplib: An aspect library for large-scale multi-core. In *Proceedings of the 42Nd International Conference on Parallel Processing*, ICPP '13, pages 270–279, Washington, DC, USA, 2013. IEEE Computer Society.

[MS13b] B. Medeiros and J. L. Sobral. Implementing an openmp-like standard with aspectj. In *Proceedings of the 3rd Workshop on Modularity in Systems Software*, MISS '13, pages 1–6, New York, NY, USA, 2013. ACM.

[MS17] B. Medeiros and J. L. Sobral. Aspect oriented parallel framework for java. In *High Performance Computing for Computational Science – VECPAR 2016*, pages 220–233, Cham, 2017. Springer International Publishing.

[MSS15] B. Medeiros, R. Silva, and J. L. Sobral. Grid programming frameworks. *Grid Computing: Techniques and Future Prospects*, pages 157–186, 2015.

[MSS16] B. Medeiros, R. Silva, and J. L. Sobral. Gaspar: a compositional aspect-oriented approach for cluster applications. *Concurrency and Computation: Practice and Experience*, 28(8):2353–2373, 2016.

[NAAL01] D. S. Nikolopoulos, E. Artiaga, E. Ayguadé, and J. Labarta. Exploiting memory affinity in openmp through schedule reuse. *SIGARCH Comput. Archit. News*, 29(5):49–55, December 2001.

[Nag06] I. Nagy. *On the Design of Aspect-Oriented Composition Models for Software Evolution*. PhD thesis, University of Twente, 6 2006.

[NB12] Marek Nowicki and Piotr Bała. Parallel computations in java with pcj library. In *High Performance Computing and Simulation (HPCS), 2012 International Conference on*, pages 381–387. IEEE, 2012.

[NC08] M. Nishizawa and S. Chiba. A small extension to java for class refinement. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, SAC '08, pages 160–165, New York, NY, USA, 2008. ACM.

[NS09] Diogo Telmo Neves and João Luís Sobral. Improving the separation of parallel code in skeletal systems. *2009 Eighth International Symposium on Parallel and Distributed Computing*, pages 257–260, 2009.

[NSPB07] James Noble, Arno Schmiedmeier, David J. Pearce, and Andrew P. Black. Patterns of aspect-oriented design. In Lise B. Hvatum and Till Schümmer, editors, *EuroPLoP*, pages 769–796. UVK - Universitaetsverlag Konstanz, 2007.

[Oak14] Scott Oaks. *Java Performance - The Definitive Guide: Getting the Most Out of Your Code*. O'Reilly, 2014.

[Obj] Introduce Parameter Object. `https://refactoring.com/catalog/introduceParameterObject.html`. Accessed: 2018-06-16.

[ONH+96] Kunle Olukotun, Basem A. Nayfeh, Lance Hammond, Ken Wilson, and Kunyung Chang. The case for a single-chip multiprocessor. *SIGOPS Oper. Syst. Rev.*, 30(5):2–11, September 1996.

[Ora18] Oracle. Collections framework enhancements in java se 8. `https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/changes8.html`, 2018. Accessed: 08-04-2018.

[Pad11] D. Padua. *Encyclopedia of Parallel Computing*. Springer Publishing Company, Incorporated, 2011.

[Par] Nested Parallelism. `https://docs.oracle.com/cd/E19205-01/819-5270/aewbc/index.html`. Accessed: 2018-06-16.

[Par72a] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[Par72b] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, December 1972.

[Par79] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, March 1979.

[PB04] E. Paul and ed. Black. Algorithms and theory of computation handbook,. In *CRC Press LLC, 1999, "single program multiple data", in Dictionary of Algorithms and Data Structures [online]*, 2004.

[PBB+06] With T. P., Joshua B., Joseph B., David H., and Doug Lea. *Java Concurrency in Practice*. Addison-Wesley Professional, Addison-Wesley, 1st edition, 2006.

[Pet11] Sanderson Peter. Updating the java grande forum benchmark suite. Master's thesis, University of Edinburgh, 2011.

[PH13] D. A. Patterson and J. L. Hennessy. *Computer Organization and Design, Fifth Edition: The Hardware/Software Interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 5th edition, 2013.

[Phe08] Chuck Pheatt. Intel® threading building blocks. *J. Comput. Sci. Coll.*, 23(4):298–298, April 2008.

[pre] Advice precedence. `https://www.eclipse.org/aspectj/doc/next/progguide/semantics-advice.html`. Accessed: 2018-06-16.

[Pre97] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Mehmet Akşit and Satoshi Matsuoka, editors, *ECOOP'97 — Object-Oriented Programming*, pages 419–443, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[PS+10] J. H. M. Pinho, J. L. F. Sobral, et al. Pluggable parallelization of evolutionary algorithms applied to the optimization of biological processes. In *Parallel, Distributed and Network-Based Processing (PDP), 2010 18th Euromicro International Conference on*, pages 395–402. IEEE, 2010.

[PSR05] R. Pawlak, L. Seinturier, and J. Retaillé. *Foundations of AOP for J2EE development*. Apress, Berkeley, CA, 2005.

[PSR13] J. Pinho, J. L. Sobral, and M. Rocha. Parallel evolutionary computation in bioinformatics applications. *Computer Methods and Programs in Biomedicine*, 110(2):183 – 191, 2013.

[Qub] Sonar Qube. Codesmell. `https://rules.sonarsource.com/java/RSPEC-1450`. Accessed: 28-03-2018.

[RC03] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *Proceedings of the 2Nd International Conference on Aspect-oriented Software Development*, AOSD '03, pages 120–129, NY, USA, 2003. ACM.

[Rei] James Reinders. `https://software.intel.com/en-us/blogs/2009/08/03/parallel_for-is-easier-with-lambdas-intel-threading-building-blocks`. Accessed: 2017-06-16.

[Rei07] J. Reinders. *Intel threading building blocks - outfitting C++ for multi-core processor parallelism.* O'Reilly, 2007.

[RHJ09] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid mpi/openmp parallel programming on clusters of multi-core smp nodes. In *Proceedings of the 2009 17th Euromicro International Conference on Parallel, Distributed and Network-based Processing*, PDP '09, pages 427–436, Washington, DC, USA, 2009. IEEE Computer Society.

[RJ14] J. Reinders and J. Jeffers. *High Performance Parallelism Pearls: Multicore and Many-core Programming Approaches.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2014.

[RM06] Awais Rashid and Ana Moreira. Domain models are not aspect free. In *In MODELS*, pages 155–169. Springer, 2006.

[SB02] Yannis Smaragdakis and Don Batory. Mixin layers: an object-oriented implementation technique for refinements and collaboration-based designs. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(2):215–255, 2002.

[SBO01] L. A. Smith, J. M. Bull, and J. Obdrzálek. A parallel java grande benchmark suite. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '01, page 8, USA, November 2001. ACM.

[SC12] N. Suri and G. Cabri. *Adaptive, Dynamic, and Resilient Systems.* Auerbach Publications, Boston, MA, USA, 1st edition, 2012.

[SCB09] Aamir Shafi, Bryan Carpenter, and Mark Baker. Nested parallelism for multi-core hpc systems using java. *J. Parallel Distrib. Comput.*, 69(6):532–545, June 2009.

[Sch01] Martin Schulz. *Shared memory programming on NUMA-based clusters using a general and open hybrid hardware, software Approach.* PhD thesis, Technical University Munich, Germany, 2001.

[SCM07] J. L. Sobral, C. A. Cunha, and M. P. Monteiro. Aspect oriented pluggable support for parallel computing. In *High Performance Computing for Computational Science - VECPAR 2006*, pages 93–106. Springer Berlin Heidelberg, 2007.

[Sema] Join Points : Appendix B. Language Semantics. `https://eclipse.org/aspectj/doc/released/progguide/semantics-joinPoints.html`. Accessed: 2018-06-16.

[Semb] Pointcuts : Appendix B. Language Semantics. `https://eclipse.org/aspectj/doc/released/progguide/semantics-pointcuts.html`. Accessed: 2018-06-16.

[SG05] M. Störzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656, 2005.

[SGS+14] Tyler M. Smith, Robert van de Geijn, Mikhail Smelyanskiy, Jeff R. Hammond, and Field G. Van Zee. Anatomy of high-performance many-threaded matrix multiplication. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 1049–1059, Washington, DC, USA, 2014. IEEE Computer Society.

[SGSP02] Olaf Spinczyk, Andreas Gal, and Wolfgang Schroder-Preikschat. Aspectc++: An aspect-oriented extension to the c++ programming language. In James Noble and John Potter, editors, *Fortieth International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, volume 10 of *CRPIT*, pages 53–60, Sydney, Australia, 2002. ACS.

[SJF+10] Hongzhang Shan, Haoqiang Jin, Karl Fürlinger, Alice Koniges, and Nicholas J. Wright. Analyzing the effect of different programming models upon performance and memory usage on cray xt5 platforms. In *Cray User's Group Meeting 2010*, Edinburgh, May 2010.

[SK10] Kotrappa Sirbi and Prakash Jayanth Kulkarni. Stronger enforcement of security using AOP and spring AOP. *CoRR*, abs/1006.4550, 2010.

[SLB02] Sergio Soares, Eduardo Laureano, and Paulo Borba. Implementing distribution and persistence aspects with aspectj. *SIGPLAN Not.*, 37(11):174–190, November 2002.

[SM07] Inc. Sun Microsystems. Sunfiretm t1000 server overview, 2007. Accessed: 2018-06-16.

[SM08] J. L. Sobral and M. P. Monteiro. A domain-specific language for parallel and grid computing. In *Proceedings of the 2008 AOSD Workshop on Domain-specific Aspect Languages*, DSAL, pages 2:1–2:4, NY, USA, 2008. ACM.

[SM11] J. Sobral and B. Medeiros. An aspect-oriented approach to fault-tolerance in grid platforms. *IBERGRID'2011 - The 5th Iberian Grid Infrastructure Conference Santander*, June 2011.

[SMC06] J. L. Sobral, M. P. Monteiro, and C. A. Cunha. Aspect-oriented support for modular parallel computing. In *Proc. of the 5th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software, Y. Coady, DH Lorenz, O. Spinczyk, and E. Wohlstadter, eds., Bonn, Germany*, pages 37–41, 2006.

[Sob06] J. L. Sobral. Incrementally developing parallel applications with aspectj. In *Proceedings of the 20th international conference on Parallel and distributed processing*, IPDPS'06, pages 116–116, Washington, DC, USA, 2006. IEEE Computer Society.

[Som10] I. Sommerville. *Software Engineering (9th Edition)*. Pearson Addison Wesley, 2010.

[Spr] SpringAOP. `http://docs.spring.io/spring/docs/current/spring-framework-reference/html/aop.html`. Accessed: 2018-06-16.

[ST15] Myoungkyu Song and Eli Tilevich. Reusing metadata across components, applications, and languages. *Sci. Comput. Program.*, 98:617–644, 2015.

[Sta12] W. Stallings. *Data and Computer Communications, Designing for performance (9th Ed.)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2012.

[SW07] Martin Sulzmann and Meng Wang. Aspect-oriented programming with type classes. In *Proceedings of the 6th Workshop on Foundations of Aspect-oriented Languages*, FOAL '07, pages 65–74, NY, USA, 2007. ACM.

[SWS12] W. Sun, X. Wang, and X. Sun. Using modular programming strategy to practice computer programming: A case study. In *2012 ASEE Annual Conference*, San Antonio, Texas, June 2012. ASEE Conferences. https://peer.asee.org/22189.

[Tan01] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2001.

[tMee] Initialize the MPI execution environment. `https://www.mpich.org/static/docs/v3.1/www3/MPI_Init_thread.html`. Accessed: 2018-06-16.

[TRE+13] Guillermo L. Taboada, Sabela Ramos, Roberto R. Expósito, Juan Touriño, and Ramón Doallo. Java in the high performance computing arena: Research, practice and experience. *Sci. Comput. Program.*, 78(5):425–444, May 2013.

[TTD03] Guillermo L Taboada, Juan Tourino, and Ramón Doallo. Performance analysis of java message-passing libraries on fast ethernet, myrinet and sci clusters. In *CLUSTER*, pages 118–126, 2003.

[TTnD12] Guillermo L. Taboada, Juan Touriño, and Ramón Doallo. F-mpj: Scalable java message-passing communications on parallel systems. *J. Supercomput.*, 60(1):117–140, April 2012.

[TUSF03] Eli Tilevich, Stephan Urbanski, Yannis Smaragdakis, and Marc Fleury. Aspectizing server-side distribution. In *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, pages 130–141. IEEE, 2003.

[UIT94] T. Utsumi, M. Ikeda, and M. Takamura. Architecture of the vpp500 parallel supercomputer. In *Proceedings of Supercomputing '94*, pages 478–487, Nov 1994.

[V.11] Michael V. Threading building blocks, openmp, or native threads? `https://software.intel.com/en-us/intel-threading-building-blocks-openmp-or-native-threads`, 2011. Accessed: 08-04-2014.

[VBP11] Ronald Veldema, Thorsten Blass, and Michael Philippsen. Enabling multiple accelerator acceleration for java/openmp. In *Proceedings of the 3rd USENIX Conference on Hot Topic in Parallelism*, HotPar'11, pages 6–6, Berkeley, CA, USA, 2011. USENIX Association.

[VDK02] Arie Van Deursen and Paul Klint. Domain-specific language design requires feature descriptions. *Journal of Computing and Information Technology*, 10(1):1–17, 2002.

[VGRGS13] Oscar Vega-Gisbert, Jose E. Roman, Siegmar Groß, and Jeffrey M. Squyres. Towards the availability of java bindings in open mpi. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 141–142, NY, USA, 2013. ACM.

[VGRS16] Oscar Vega-Gisbert, Jose E. Roman, and Jeffrey M. Squyres. Design and implementation of java bindings in open mpi. *Parallel Computing*, 59(Complete):1–20, 2016.

[VGS13] Vikas, Nasser Giacaman, and Oliver Sinnen. Pyjama: Openmp-like implementation for java, with gui extensions. In *Proceedings of the 2013 International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM '13, pages 43–52, NY, USA, 2013. ACM.

[VHBB01] R. Veldema, R. F. H. Hofman, R. A. F. Bhoedjang, and H. E. Bal. Runtime optimizations for a java dsm implementation. In *Proceedings of the 2001 Joint ACM-ISCOPE Conference on Java Grande*, JGI '01, pages 153–162, NY, USA, 2001. ACM.

[Vin99] B. Vinter. *PastSet - A Distributed Shared Memory System*. PhD thesis, Tromso Univ. , Norway, 1999.

[VV17] Mikael Vidstedt and Sandhya Viswanathan. Jdk 9 hidden gems [con4529]. `https://events.rainfocus.com/catalog/oracle/oow17/catalogjavaone17?search=CON4529&showEnrolled=false`, October 2017. Accessed: 01-04-2018.

[Wal91] David W. Wall. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, pages 176–188, NY, USA, 1991. ACM.

[Wam06] Dean Wampler. The Challenges of Writing Reusable and Portable Aspects in AspectJ: Lessons from Contract4J. In *In Proceedings of International Conference on Aspect Oriented Software Development (AOSD 2006)*, March 2006.

[Wea] Load-Time Weaving. `https://eclipse.org/aspectj/doc/released/devguide/ltw.html`. Accessed: 2018-06-16.

[Wei] Eric W. Weisstein. Fourier series. `http://mathworld.wolfram.com/FourierSeries.html`. Accessed: 28-03-2018.

[WPH03] P. Werstein, M. Pethick, and Z. Huang. A performance comparison of dsm, pvm, and mpi. In *Proceedings of the 4th International Conference on Parallel and Distributed Computing, Applications and Technologies*, pages 476–482, SW Jiaotong University, Chengdu, China, 2003.

[XHG09] C. Xi, B. Harbulot, and J. R. Gurd. Aspect-oriented support for synchronization in parallel computing. In *Proceedings of the 1st Workshop on Linking Aspect Technology and Evolution*, PLATE '09, pages 1–5, New York, NY, USA, 2009. ACM.

[YBC+07] K. Yelick, D. Bonachea, W. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen. Productivity and performance using partitioned global address space languages. In *Proceedings of the International Workshop on Parallel Symbolic Computation*, PASCO, pages 24–32, NY, USA, 2007. ACM.

[YC79] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1st edition, 1979.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, and Alex Aiken. Titanium: A high-performance java dialect. In *In ACM*, pages 10–11, 1998.

[ZMWK17] Zhengji Zhao, Martijn Marsman, Florian Wende, and Jeongnim Kim. Performance of hybrid mpi/openmp vasp on cray xc40 based on intel knights landing many integrated core architecture. In *CUG Proceedings*, 2017.