Locality optimisation techniques for platforms

Rui Silva

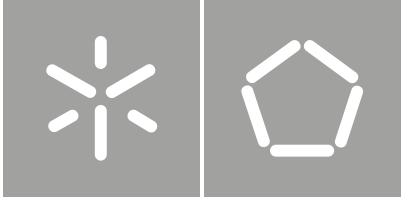
Uminho | 2021

Universidade do Minho
Escola de Engenharia

Rui António Sabino Castiço da Silva

Locality optimisation techniques for platforms

april 2021



Universidade do Minho
Escola de Engenharia

Rui António Sabino Castiço da Silva

Locality optimisation techniques for platforms

Doctor Program in Informatics

Supervisors:
Professor João Luís Ferreira Sobral

april 2021

Despacho RT - 31 /2019 - Anexo 3

Declaração a incluir na Tese de Doutoramento (ou equivalente) ou no trabalho de Mestrado

DIREITOS DE AUTOR E CONDIÇÕES DE UTILIZAÇÃO DO TRABALHO POR TERCEIROS

Acknowledgements

Obtaining a doctoral degree comes with a personal decision to improve my knowledge of high-performance computing. My view is that performance should not interfere with the domain code. During this work, techniques were explored in order to expose this vision On this long journey, I counted with many individuals and institutions help, which made it possible to obtain the final results. First of all, I want to deeply thank my advisor, Professor Doctor João Luís Sobral, who always supported me, sharing his vision, asking pertinent questions that made it possible to choose the best solutions. Secondly, I would like to thank my research colleagues who had always been willing to collaborate and share ideas. To Bruno Medeiros for the different collaborative works presented throughout this dissertation. To Rui Gonçalves for the long conversations about the possible implementations and the constraints of each solution. I would like to thank the Informatics Department for the working conditions provided and for access to the Search-ON2: Revitalization of HPC infrastructure of UMinho, (NORTE-07-0162-FEDER-000086), co-funded by the North Portugal Regional Operational Programme (ON.2-O Novo Norte), under the National Strategic Reference Framework (NSRF), through the European Regional Development Fund (ERDF). The project Parallel Refinements for Irregular Applications (UTAustin/CA/0056/2008) and the project General-purpose Aspect-Oriented framework for heterogeneous multicore Parallel systems (PTDC/EIA-EIA/108937/2008), both supported the first years of this thesis. Finally, I would like to thank all the people who were part of my life during this journey. To my parents who, in difficult times, were there and encouraged me to continue. My sister shared many moments of my life in this period. To my friends Jorge and Andreia who were always available to help. Finally, Isabel, who appeared in my life in the final phase of this marathon and who encouraged me every day to finish it.

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, _____

Full name: _____

Signature: _____

Resumo

Aplicações científicas simulam o mundo real através de modelos matemáticos. As simulações destes modelos necessitam de grande poder computacional, existente nas arquiteturas atuais. Contudo, para aceder a esse poder computacional, o programador necessita de desenvolver a aplicação de acordo com a plataforma de execução, o que introduz complexidade no desenvolvimento da aplicação. Nas abordagens tradicionais estas adaptações/otimizações estão misturadas no código do domínio originando dois problemas: primeiro, o código fica dependente da plataforma, sendo que a execução numa plataforma distinta obriga a uma reescrita do código; segundo, o código relativo à otimização mistura-se com o código do domínio, dificultando a perceção do mesmo.

As linguagens orientadas ao objeto são reconhecidas por explicitar os conceitos do domínio no código. Porém, a sua utilização introduz tipicamente uma elevada sobrecarga na execução da aplicação limitando a sua utilização em aplicações cientificas. Isso explica o motivo pelo qual o Java, uma das linguagens orientadas ao objeto mais utilizadas, não é usada neste tipo de aplicações. Java utiliza a compilação dinâmica para remover as sobrecargas das linguagens orientadas ao objeto, quando os conceitos mais avançados não são utilizados (exemplo polimorfismo), como é, frequente, no caso das aplicações científicas.

A abordagem apresentada nesta tese permite adiar a implementação das otimizações para fases posteriores do desenvolvimento, escondendo o mapeamento de dados. Por outro lado, permite especificar nas fases finais do desenvolvimento várias optimisações: o processamento em subdomínios, empacotamento de dados e ordenação dos dados em memória. Por fim, permite a execução paralela ocultando detalhes de implementação. Para isso separa o desenvolvimento em duas fases distintas: escrita do código de domínio e fase de otimização. A fase de otimização é adiada para fase final do desenvolvimento, o que permite uma fácil adaptação à plataforma de execução.

A abordagem permite aplicar estas otimizações através de dois mecanismos: primeiro, alteração do mapeamento das coleções; e segundo, decomposição do problema em subproblemas. Ambas as otimizações são introduzidas no programa pelo programador de uma forma simples (pequeno custo de desenvolvimento) mantendo os conceitos de domínio. Primeiro, a alteração do layout baseia-se no conceito de procurador, cria um objeto temporário o que permite ao utilizador usar vários mapeamentos com a mesma API. Em segundo lugar, o mecanismo de decomposição de domínio suporta outras otimizações comuns: processamento de dados em blocos, empacotamento, execução paralela e dados privados aos fios de execução. O mecanismo é implementado por anotações de código, evitando alterações mais invasivas.

A abordagem foi avaliada com um conjunto de casos de estudo: soma de um vector, daxpy, JECoLi, simulação de dinâmica molecular e multiplicação de matrizes. Este conjunto permitiu validar a abordagem em diferentes casos e a sobrecarga introduzida na execução. A adaptação do código de domínio para suportar a abordagem foi mais simples do que alterar o mapeamento de dados no código de domínio. Em todos os casos, a abordagem obteve uma performance similar às abordagens tradicionais. No caso do MD, exemplo que suporta mais otimizações, o uso da abordagem proporcionou um ganho de 50X no tempo de execução. Os outros casos de estudos obtiveram ganhos entre 20x e 40x. A JECoLi teve um ganho mais baixo (1,6x), já que neste caso apenas foi possível aplicar a otimização do mapeamento dos dados. Esses ganhos mostram a viabilidade da abordagem que permitiu obter códigos eficientes.

Palavras chave: Execução paralela, Java, organização de dados, otimizações para hierarquia de memória, *Tiling*

Abstract

Scientific applications simulate the real world through mathematical models. The simulation of these models requires all the computational power available in current architectures. However, to take advantage of this computational power, the simulation code must be optimised, bringing it closer to the execution platform. This adaptation introduces a lot of complexity in application development. Traditionally the code is written according to the platform on which it will be executed. This approach has two problems: first, the code is dependent on the execution platform, and changing the execution platform requires code rewriting; second, the optimisation code is mixed with the domain code, making it difficult to understand.

The Object-Oriented Paradigm (OOP) is known for bringing the code closer to the domain. However, its use typically introduces an overhead, preventing its use in scientific applications. This explains why Java, one of the most widely used OOP languages is not commonly used to develop scientific applications. On the other hand, modern OOP languages that rely on dynamic compilation (e.g., Java) can remove many overheads typical of OOP, when more advanced features are not used (e.g. polymorphism), which is the case of many scientific applications.

This dissertation introduces an approach that allows developers to perform optimisation in the final development step. The approach enables multiple data layouts and allows the selection of the best layout according to the execution platform. On the other hand, the approach supports tiling, packing and sorting optimisations. Additionally, the approach supports parallel execution, hiding the implementation details related to optimisation. The approach separates the development of the domain code from its optimisation. The optimisation step is delayed until the final development step, which allows an easy adaptation to the execution platform.

The supported optimisations rely on two mechanisms: first, changing the data collection layout and second, decomposing the problem into subproblems. Both optimisations are introduced in the program by the developer in a simple way (low development cost) and maintaining the domain concepts in the code. First, for hiding the data layout, the approach is based on the proxy pattern, creating a temporary object that accesses the data using the same API. Second, the domain decomposition mechanism enables several common optimisations: processing data in tiles, packing, parallel execution and thread private data. The technique was implemented by code annotations avoiding more invasive code changes.

The approach was evaluated with a set of case studies: Sum, daxpy, JECoLi, MD and Matrix multiplication. This set allowed to verify the approach effectiveness in different cases and its execution overhead. The adaptation of the domain code to support the approach was simpler than transforming the layout in the domain code. In all cases, the approach obtained a performance similar to traditional approaches.

In the MD case, the example that supports more optimisations, the use of the approach provided a gain of 50x in execution time. Other cases studies provided gains from 20x to 40x. The JECoLi case has the lowest gain (1.6x) since the gain was only due to layout change. These gains show the feasibility of the approach that delivered efficient optimised codes, adding low additional cost when compared with traditional approaches.

Keywords: Data layout, Java, Optimisations for memory hierarchy, Parallel execution, Tiling

Contents

List of figures

List of listings

List of tables

Chapter 1

Introduction

1.1 Context

Scientific applications simulate real-world activities using mathematical models, converting the real-world entities and activities into computational models. This kind of applications requires massive computational power to perform accurate simulations. Frequently, a set of optimisations is needed to efficiently use the computational power available. Optimisations make the development of scientific applications more complex: first, the developer needs knowledge of the domain and execution platforms; second, the optimisations must be applied depending on the execution platform (platform map). Sometimes the developer uses libraries to obtain high performance, but the solution cannot be implemented in all cases.

Object Oriented Programming (OOP) languages support more complex codes by using objects to represent entities in the domain (i.e. real-world entities). The objects hold the state of those entities [SB85, Weg87] and have methods that are actions over the entity. Thus, OOP languages enable the development of more abstract code due to the usage of domain concepts, making the code more perceptible [Sny86]. It leads to easier programming and fewer errors. In OOP, the encapsulation concept is a key to provide abstraction: it defines that objects must have methods to access them because their internal representation should not be relevant.

Java is currently one of the most popular OOP languages. It can be used to develop scientific applications due to its flexibility and provides fast development. Java provides a set of containers (Java Collections Framework (JCF)) that helps develop applications. These containers are organised by type, where the top of the hierarchy is divided into collections and maps. This sort of hierarchy is made up of interfaces, providing multiple implementations. Moreover, the developer can provide new implementations if the built-in implementations are not appropriate for the case (e.g. low performance). It is accomplished by developing

new containers that conform to the JCF data access interface.

JECoLi [EMR09] is one good example of a framework to develop scientific applications in Java. It provides a large set of generic algorithms that the developer can use for the fast development of applications in this scientific domain. OpenFOAM [JJT+07] is another example in the C++ world that use OOP to simplify programming. However, Java is simpler than C++, more secure and leads to more robust programs [YSP+98]. On the other hand, it is more challenging to develop efficient applications in Java.

The proliferation of multicores makes computational power compatible with the scientific application. The wide availability of multicore systems was mostly motivated by two factors [MRR12]: energy limitation and parallelism limitation at the level of instruction. The energy limitation restricts the maximum processor frequency mainly due to the heat dissipation limits. On the other hand, improving the exploitation of parallelism at the instruction level became limited due to the lack of instructions without dependencies in most programs. The solution came with multicores proliferation and the need to use parallel programming to increase performance.

The effective use of multicores systems requires the execution of code in parallel. Specifying parallel execution improves performance but also increases the code complexity [PGB+06]. With the introduction of multicores, the shared memory model became widely used. In the shared memory model, all threads access global memory. Different tools have been created to help parallel applications development under this model. This thesis will highlight OpenMP and Java Streams as the two most predominant parallel programming environments for multicores. OpenMP uses the fork-join model specified through directives in C, C++ and Fortran. Java 8 recently introduced streams that provide parallelism at the data-level. Data-level parallelism applies the same instructions (block of code) to different data in different processing units. It is the most common form of parallelism in scientific applications. The Java streams provide a simple and secure specification of data-level parallelism.

The processor-memory performance gap has increased over the last decades [Car02]. The processing units improved in computational capacity, but their real performance became constrained by the memory access limitations [LRW91]. It became fundamental to introduce a memory hierarchy to mitigate the problem. The memory hierarchy improves the data access performance, lightening the main memory accesses, namely by getting blocks instead of single words. Modern processor units can take advantage of spatial locality (access to data whose memory location is close to previous access), and temporal locality (access to data that has been accessed recently). Therefore, in many cases, the developer needs to follow new programming rules in order to improve temporal and/or spatial locality. There are well-known techniques to improve data locality. One common technique uses a more efficient data layout in collections but makes the code less abstract.

Typically, scientific applications need a large data set. These sets are represented as collections of objects (domain entities), which impact the program performance depending on its data organisation in memory. The collections may use one of three distinct layouts: AoP, AoS and SoA [NFS11]. AoP is an array of pointers, where each datum is allocated in a new memory region, which requires more memory accesses. However, this layout natively supports polymorphic data in C++ and Java (collections can hold entities of different kinds). The AoS layout improves performance by reducing the number of memory accesses (access to the element implies a calculation of its position instead of getting a memory address). In SoA, the data structure is broken into multiple parts, creating several arrays. To summarise, typically, more abstract layouts tend to decrease performance, so one pragmatic data locality optimisation in scientific codes is the use of SoA layout.

There are other well-known memory locality optimisations. Kowarschik [KW03] organises optimisations in two groups: data organisation and reordering accesses to data. The data organisation intends to reduce cache line conflicts and increase spatial locality. These optimisations change the memory footprint, so the elements are still processed in the original (sequential) order. On the other hand, the accesses reordering changes the order that elements are accessed to increase the temporal locality. Stratton [SRS+12] proposes a similar classification for massive thread systems optimisations. In data organisation, packing and sorting are commonly used. The packing compacts data required for processing. Sorting orders elements by the order they are accessed. As for accesses reordering, the most common optimisation is tiling. In this case, the processing is performed in small parts to keep the data in cache.

1.2 Motivation

The development of efficient programs for today's processing units (e.g., multicore systems) requires the efficient use of the memory hierarchy. Current compilers are not able to automatically perform this task due to the complexity and interplay of memory locality optimisations. In many cases, the developer must ensure the correctness and effectiveness of optimisations. For example, changing the layout from AoS to SoA forces a change in the signature of the function/method requiring several transformations in the code. Moreover, the effectiveness of many locality optimisations depends on the execution platform. Frequently, many of these optimisations are implemented in the premature steps of the code development [Str11], making the code more complex [Oak14] since the domain concepts are removed or overshadowed by the optimisation code. The challenge is how to develop high-level code (e.g., abstract) and how to optimise the code in the final development step (e.g., by creating efficient platform mappings).

Scientific applications traditionally use low-level languages such as C and Fortran. These languages

allow the writing of code closer to the execution platform but have less support to develop abstract codes (e.g., using the domain concepts in the code). An example is specifying the memory management directly in the code that limits the runtime optimisations (e.g., locality of data accesses). On the other hand, Java uses the garbage collector to manage memory and compiles the code at the runtime (dynamic compilation). We believe that writing abstract code simultaneously with dynamic compilation will help to apply optimisations in the final development step and will deliver acceptable performance.

Java is a language that abstracts the execution platform bringing the code closer to domain concepts. However, Java is criticised for introducing additional execution overhead. Some mechanisms that introduce overheads are the Garbage Collector, execution on a virtual machine and polymorphic collections. In scientific applications, it is common to use the same data set during the simulation making the Garbage Collector cost negligible. On the other hand, it is also common to execute the same procedures multiple times that enables the virtual machine to dynamically compile the code generating efficient code for the native platform. Additionally, the dynamic compilation allows applying additional optimisations. The polymorphic collections are not essential in this type of applications. We believe the Java mechanisms make it is feasible to develop efficient scientific applications in Java.

1.3 Objectives

Within the scope of this dissertation, we intend to develop a new approach to improve the programmability of scientific applications without dismissing the performance. The approach should support the most common locality optimisations without removing or overshadowing the domain concepts.

This work aims to achieve the following three specific objectives:

- Develop an API compatible with Java Collections that efficiently maps on the execution platform;
- Develop a high-level constructor that supports tiling and parallelism;
- Study the support for other locality optimisations commonly used.

The API should support OOP concepts and provide efficient layouts (e.g., SoA). The API will be compatible with the Java Collections simplifying the application development. The most efficient layout is typically the SoA layout, where a collection becomes a structure of several arrays. Usually, this removes the object abstraction from the code and a change in the layout forces the developer to perform broad changes in the code. The usage of OOP concepts with the SoA layout and maintaining compatibility with the Java API is the main challenge of this dissertation.

The high-level constructor will support tiling and parallelism. The constructor will support problem processing by decomposing the domain into smaller problems, enabling parallel processing of subdomains, tiling optimisation or both. The high-level constructor should hide the details of those optimisations. The developer should access only the relevant parameters, such as the number of threads of execution. It must be possible to combine several levels of this optimisation.

The last objective is to study and implement other common optimisations compatible with the developed mechanisms. For example, tiling can use packing, the data tile is copied into a new data structure, processing elements placed in consecutive memory locations.

1.4 Contributions

The contributions of this work are as follows:

1. An approach to develop abstract code and to optimise the code at final the development step;
2. An approach that encapsulates the data layout and allows the layout selection in compile/execution time;
3. A high-level constructor inspired on the map-reduce pattern that can express both parallel computations and tiling over a data collection;
4. Several optimisation techniques applied to a Java-based implementation of the proposed data API and parallelism model;
5. A set of tools to support the approach;
6. A benchmark with scientific applications.

The main contribution of this work is a new approach to develop scientific applications separating the development into two distinct steps: domain, and optimisation. In the domain step, the methodology provides an API to simplify the development of abstract code. The API enables the data layout optimisation in the final development step (contribution 2). The approach also enables other improvements in the optimisation step. It offers a high-level constructor to inject other supported optimisations (contribution 3). The developed constructor is based on a map-reduce pattern, but it also supports tiling, packing and privatisation (contribution 4). Additionally, the constructor uses Java annotations to simplify the optimisation specification with minimal impact on the domain code (avoid overshadowing the domain concepts).

During the development of this work, a toolset was developed to support the approach. The first tool generates collections with two different layouts: AoP and SoA. The developer specifies the domain model, and the tool generates collections implementations. The collections provide all methods to access the object and higher-order functions to support other approach features. A second tool simplifies the parallelism specification. For this, the developer creates an annotation for the method that defines: the domain decomposition; how each subdomain is processed; and how to join the subdomains. Finally, a performance analysis tool, based on aspects, provides performance counters.

A study using a set of benchmarks provided a validation of the approach. These benchmarks allow the performance study of the Java language in scientific applications. In some cases, the assembly code is analysed to show the efficiency code generated in Java. Java collections and streams were analysed. Finally, the benchmark clarifies the reasons for the performance differences obtained with traditional approaches versus the approach presented on this thesis.

During the development of this work, these contributions were disseminated in the scientific community. The data API was presented in the "Gaspar: A Compositional Aspect-Oriented Approach for Cluster Applications" [MSS15]. In VecPar [SS16], we present API improvements, and introduced the a high-level constructor. An early study of the layout impact on performance was published in [FSS13].

In this work, the acronym *GasPar* refers to the implementation of our approach. The acronym comes from a project that explored the separation of concerns (i.e., domain and optimisation code) for the different platforms. Our approach comes within the scope of the project with a focus on optimising data access. In parallel, other work focused on parallel execution through aspects [Med19].

1.5 Outline

The document is organised into five chapters. The next chapter (chapter 2) presents the important concepts for the approach development. It describes the memory hierarchy, the common memory optimisations and presents the Java language and execution environment. The memory hierarchy section shows details of the operation of cache levels. The second section presents the main optimisations of data access. And finally, the last section introduces the differentiating characteristics of the Java environment.

Chapter 3 describes the approach developed in this work. First, we define the problem and requirements of the approach. Subsequently, we show how it satisfies the requirements. The chapter also describes the programming interface. Next, the chapter describes the implementation of the approach. Finally, the chapter explains the most relevant imitations.

Chapter 4 describes the evaluation of the approach presented in chapter 3, applying optimisation to

several case studies. The first case uses two simple algorithms to analyse the basic cost of the programming abstractions. The next evaluation uses the generic framework JECoLi to test the approach on existing code. The third case study uses a molecular dynamics simulation(MD) to test the approach with more complex structures. The last case is a matrix multiplication that tests the framework extensibility with a new container. The chapter ends with a summary of the results and presents the conclusions.

Chapter 5 discusses alternative approaches to the one described in this thesis. It starts to argue about iterators and then it describes, for each optimisation presented in section 2.2, what are the existing alternatives and what their characteristics are. Finally, it compares the alternatives described with the approach proposed. The last chapter (chapter 6) draws conclusions and presents research proposals to be developed in the future.

Chapter 2

Background

This chapter introduces the main concepts used in this work. First, we describe the memory organisation and its impact on performance. Subsequently, the main optimisations used to improve data access are presented. At the end of the chapter, we describe the Java characteristics.

2.1 Memory hierarchy

The gap between CPU frequency and the time to access data in memory has been increasing over the last decades [Car02, MCWK99]. The data access performance is characterised by bandwidth and latency. The bandwidth is the data amount that can be transferred per unit of time. Latency is the time it takes to get the first data bit. The problem in accessing data lays in the latency that has not kept up with the processor speed. Introducing cache memory reduces the negative impact of the latency on performance, but only when programs provide locality in data accesses. These caches reduce the cost of data accesses due to: the most used data are in faster memories (temporal locality), or the data are in consecutive positions in memory (spatial locality with pre-fetch). Sometimes, more data is transferred than necessary (some data are not used), which consumes more bandwidth. Double Data Rate memories emerged, which allowed the bandwidth to double, but they did not reduce the latency. The use of caches improves the average latency but consumes more bandwidth. Thus, it is possible to have faster data accesses when these are stored on consecutive memory addresses.

The cache memory also improves the data accesses performance when the data are already in the cache. Data may not be available in the cache due to four factors [SJG92, SHV+98]: cold, capacity, conflicts and coherence. Cold is when data has not ever been loaded into the cache memory. Capacity is when the amount of data exceeds the cache memory capacity, and some data must be discarded from

the cache. Most cache memories have a set of spaces for each memory address, which might originate a conflict miss when the set is full and more data is loaded to this set. Conflicts and capacity misses reduce if the cache capacity increases or the data footprint decreases. Coherence misses occurs when two threads access the same memory address, and one of them writes the data and to maintain coherence, the data from the processor/core are invalidated. It can also happen when threads write at nearby addresses since the control is, generally, carried out on a cache line basis. The problem is known as false sharing.

Figure 2.1 shows a typical memory hierarchy. The top-level registers have faster access time (e.g., as fast as the machine clock time), but their capacity is too much limited (e.g. 32 registers). Cache memories reduce the performance gap between the main memory and the processor. The main memory can be extended using virtual memory, but it has a longer access time. The remainder of the section describes how cache memory works.

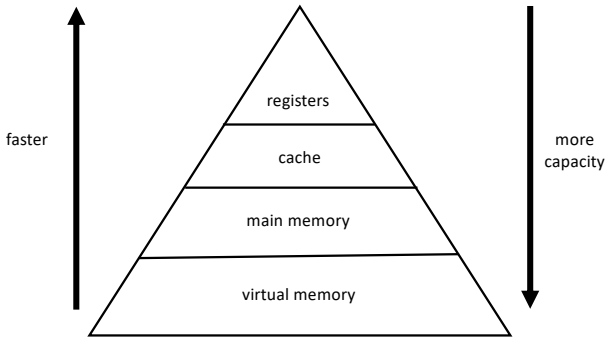


Figure 2.1: Typical memory hierarchy

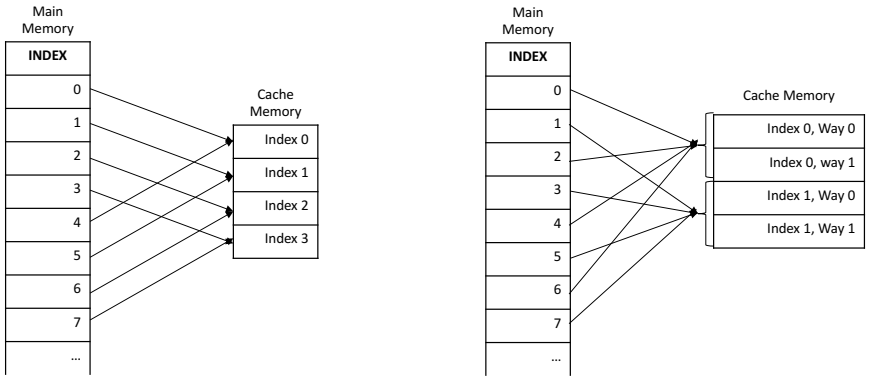
Table 2.1 shows the access time and the size of the different memories. These memories are temporarily used to keep a copy of data from the main memory. There are some policies to manage these temporary data.

Memory	**Latency**	**Size**
L1	4 cycles	32 + 32KiB/per core
L2	10 cycles	256KiB/per core
L3, line unshared	40 cycles	
L3, shared line	65 cycles	2MiB x #cores
L3, modified in	75 cycles	
L3, remote	100-300 cycles	
Local Dram	60 ns	$>$ GiB
Remote Dram	100 ns	$>$ GiB

Table 2.1: Access time and size the memory hierarchy for Intel Xeon 5500 processors [Lev09]

The memory bottleneck can be a big hurdle in multicore platforms since the latency does not decrease, and the cores share the available memory bandwidth. On the other hand, the effective bandwidth in accessing caches scales proportionally to the number of cores, since most platforms provide an L1/L2 cache per core. Thus, to effectively use, current and future, programs should explore temporal and/or spatial locality in data access whenever possible. The next paragraphs describe how the data are mapped in the cache, which data stay in the cache when more data is loaded from the main memory and the protocols to keep these memories with coherent values.

There are different options to map a block of main memory in the cache lines: *Direct Map*, *N-Way Associative Map* and *Fully Associative Map* (figure 2.2). In the *Direct Map*, the memory address is mapped into one cache line (figure 2.2a). This implementation is simple and inexpensive, but it has the highest conflict misses. In the *Fully Associative Map*, each address can be mapped to any cache line. Therefore, there is a better performance since there are no conflict misses (better cache usage). However, this mapping has the highest hardware cost. *N-Way Associative Map* (figure 2.2b) is more flexible than *Direct Map* and easier to implement than *Fully Associative Map* . It allows each memory address to be mapped to N different cache lines (for a set).



(a) Direct Map (b) Associative map

Figure 2.2: Cache maps

The N-Way and Fully associative maps need a protocol to select the data to be replaced (i.e., dismiss). A simple protocol replaces the older block in the cache. The implementation can be done using a circular buffer to indicate where the next block is placed. On the other hand, there are more complex protocols: Least Recently Used (LRU), Least Frequently Used (LFU) and Not Recently Used (NRU). In LRU, the block replaced is the block that was accessed less recently. The LFU replaces the block that has less usage. Both protocols require an additional state to save the usage of the blocks. It is impracticable to use the

LRU. For this reason, the modern processors use the NRU, which discards a block not accessed recently [JTSE10]. NRU uses one bit per data block that indicates if the block can be discarded or not. When it needs to discard a block, it firstly selects one of these marked blocks. These protocols are relevant for programming techniques that improve the temporal locality since the set of data that is accessed (most) recently is the one that remains in the cache.

The memory caches can implement different data write policies (what and when the different hierarchy levels are written). There are two different policies: write-through and write-back. In the write-through, when data is written it is immediately also written in the main memory. The write-though guarantees coherence with the main memory. Because the data is always written to memory, write operations are slower and consume more memory bandwidth. The other alternative, write-back, reduces the writes on main memory to the minimal (for example, when the cache is full, and it needs to replace some data). However, this alternative makes the data coherence protocols more complex. Performance improves in two ways: reduces bandwidth consumption, and the writes are in the cache memories (faster writers). On the other hand, data readings can cause a write miss if a line should be replaced it needs to write in the main memory. Current processors use write-back, so the write miss is not a key issue for performance.

In modern multicores, each core has its top-level cache(s), it is necessary to ensure that the values in the memory hierarchy are coherent (i.e., all cores see the same value). Typically the control of data coherence is done at cache line level (if an element of a line changes in one core, the line must be made coherent in all cores). For this, the cache memories use state information for each line. The available states depend on the protocol used.

One of the most naive protocols is MSI. In this protocol, the cache line has three possible states: Modified, Shared and Invalid. In the *Modified* state, the value has changed, and the cache will write the new value in memory. If the line is in *Shared* state, the line was only read, and the value can be used. The *Invalid* state identifies that the line has changed by another processing unit and needs to be updated. From this cache coherence protocol emerges the MESI protocol that adds the *Exclusive* state. This new state identifies which blocks are loaded into only one cache memory of a processing unit. Intel processors use the MESIF protocol that derives from MESI and adds a *Foward* state. The *Forward* state is a specialisation of state *Shared*. When a processing unit changes a block of data, that block is marked in state *Foward* in that cache. From that moment, the remaining cache updates are performed using the value from that cache. At each moment and for each block there is only one core with state *Foward*.

Current processors can load data into the cache memory before it is needed, this technique is called pre-fetch. The processors predict the data which will be used by the next instructions and load that data into the cache memory. The pre-fetch improves performance, especially if the data access is sequential

(e.g., sequential accesses to positions of an array). However, pre-fetch increases bandwidth consumption and creates additional cache line conflicts. If the program accesses the data randomly, pre-fetch may cause a degradation of performance.

On the other hand, the processors provide pre-fetch instructions that allow the developer/compiler to identify the data which will be used later. When the data are marked for pre-fetch, the data pattern detection is disabled. Therefore, there is a decrease in bandwidth consumption, as the pattern detector does not load additional data.

2.2 Data locality optimisations

The previous section presented the relevant characteristics of the memory hierarchy. This section presents the most used optimisation techniques [KW03, SRS+12] that can take advantage of these characteristics in scientific applications. The techniques can be grouped into how data are stored in memory (Data footprint) and the order in which data are processed (Data flow). The section starts with techniques that modify the footprint of data in memory. The subsequent section addresses techniques that change the processing flow. Finally, the section ends with a summary of all techniques described.

2.2.1 Data footprint

This section shows the main optimisations that modify the data footprint in memory. The first optimisation changes the layout to take advantage of the memory hierarchy. The second optimisation sorts data in memory to improve spatial locality. Next, padding and alignment are presented, which create empty spaces to make data access more efficient. Finally, the packing optimisation compacts/organise data during execution to maximise the cache usage.

Data layout

The data can have different layouts in memory. The most common data layouts for a collection of objects (or data structures) are (figure 2.3): Array of Pointers (AoP), Array of Structures (AoS) and Structure of Arrays (SoA). AoP and AoS use an API closer to real-world entities (e.g., developers can work with data structures instead of array indexes) [JRS16].

The AoP layout is a popular layout due to its support for abstract data types. The collection is an array of pointers to the concrete data type. The other layouts improve spatial locality by storing data in contiguous memory addresses.

In AoS, the entities are stored in contiguous memory addresses, as in SoA, which stores fields into separate arrays. The AoS layout removes the array of pointers and uses an array of structures (the data is contiguous in memory). This layout is not available in Java. The SoA provides better locality if the algorithm does not require all structure fields in the same time-frame, loading only the required fields.

The AoP requires additional space to hold the array of pointers, when compared to SoA, but provides more flexibility to manage the data storage. It is beneficial to use hybrid layouts, in certain situations, where some fields are stored in the contiguous memory address, the remaining are stored into separate arrays. Sharma [SKK+13] explores these hybrid layouts.

The data layout can have a significant impact on performance, and the choice of the best might depend on the platform and algorithm [MBZ+13, SKK+13]. Moreover, the change from one layout to another might require considerable code refactoring.

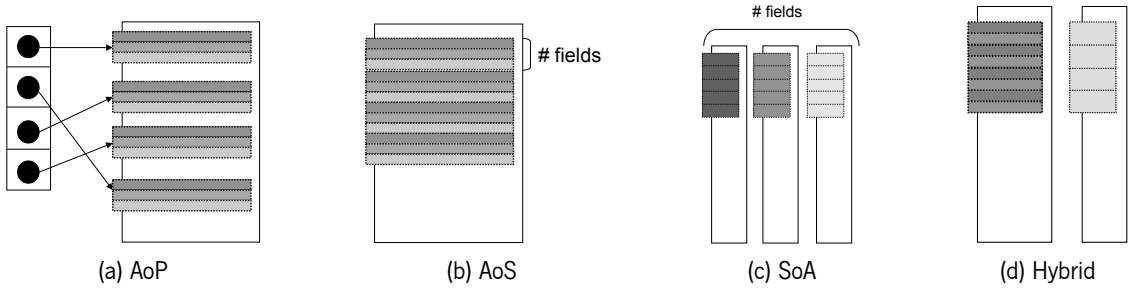


Figure 2.3: Layouts footprint in memory

One concrete example is discussed next. In the listing 2.1, to change from the AoP layout to the SoA or the AoS the *computeForce* must be changed accordingly: creates additional parameter (change API), and adapt code implementation. Moreover, the AoP layout is the most natural implementation as the code includes a concept from the domain application (*Particle*). The *Particle* has a set of methods to implement actions in the *Particle* (e.g., *computeForce*). The change from this layout to other requires rewriting code in all places where the *Particle* class is used in the code. In the Java Grande Forum[1] (JGF) MD used in this document, a change in the layout implies an extensive code rewrite. The AoS layout maintains the entity but needs to pass the *Particle* by parameter. Moreover, Java does not support this layout but, in this case, can emulate this layout. However, this solution removes entity representation from the code.

To access the *px* particle field in the array, each layout requires:

AoP two memory accesses to access the element;

[1] `http://www.javagrande.org/`

```
//AoP layout
forceParticle(...){
  xi = this.px;
  yi = this.py;
  zi = this.pz;
  (...)
}
```

```
//AoS layout
forceParticle(Particle p1, ...){
  xi = p1.px;
  yi = p1.py;
  zi = p1.pz;
  (...)
}
```

```
//SoA layout
forceParticle(Particles p1, int id, ...){
  xi = p1.px[id];
  yi = p1.py[id];
  zi = p1.pz[id];
  (...)
}
```

```
//AoS layout in Java
forceParticle(Particles p1, int id, ...){
  xi = p1.data[id*9+0];
  yi = p1.data[id*9+1];
  zi = p1.data[id*9+2];
  (...)
}
```

Listing 2.1: Force computation using AoP, AoS and SoA layouts.

AoS one memory access and a calculation to access the data;

SoA one memory access to access data.

The SoA implies fewer instructions. However, sometimes other layouts can be more efficient. In the AoP layout, to swap the data, it is possible to switch the pointers. However, in AoS and SoA, it needs to copy field by field. In AoS, all fields have a better spatial locality than SoA (each entity has all data in contiguous memory positions).

Sometimes the developer needs to test several layouts to obtain the best performance. Traditionally there are two approaches to change the layout: rewrite code or encapsulate the data. The need to rewrite the code creates a code maintenance problem (requires more code, one for each layout). Encapsulating the data can have an impact on performance [FSS13].

The data layout optimisation is equivalent to combination of *Merge data structures* (Array Merging) and *Transpose* [KW03, SRS+12]. The *Merge data structure* optimisation groups the structures and changes the SoA layout to the AoS layout. The *Transpose* changes AoS to SoA or from SoA to AoS.

Data layout in Java collections

Java has the Java Collections Framework (JCF) that supports a set of containers. It provides different ways to store the data with the same behaviour. Containers manage the memory without developer intervention and allow hiding the collection layout (e.g., *ArrayList* vs *LinkedList*). All containers available in the JCF support generics. This forces collections to use the AoP layout, making the default JCF implementations not suited for HPC.

Figure 2.4 shows the JCF class hierarchy. JCF provides a set of interfaces, and collections implementations organised hierarchically. At the top of the hierarchy, there are two interfaces, *Collection* and *Map*.

The collection interface represents a collection of objects, that can be ordered or not and with or without repeated elements. There are three interfaces that extend the collection interface: *List*, *Queue* and *Set*. In the *List*, there is a specific elements order. The elements are associated with the order they occupy in the collection, being possible to access them through their position (*get(i)*). *Queue* adds to the collection a set of methods that make it possible to add and remove elements from the container. The elements are removed according to their insertion order. In the *Set*, there is no order of the elements, and it does not support repeated elements.

The *ArrayList* is the most used data structure of the JCF, approximately 47% in the study by Costa [CASL17]. The OpenJDK implementation uses an array of pointers. Additionally, the class adds a mechanism that allows the developer to add and remove elements without overloading the developer with the array dimension management. The *Stack* class implements the *List* interface and adds methods that allow the developer to use the container as a *Stack* (the last element inserted is the first to be removed). *LinkedList* implements the *List* and *Queue* interface at the same time. Its internal structure is based on nodes, where each node has the pointer to the next node and the other to the previous element. *PriorityQueue* implements the *Queue* interface using an array organised as binary heap.

The *Map* interface represents a set of elements in which each element has an associated key. In the JCF there are two implementations of the *Map* interface: *HashMap* and *TreeMap*. *HashMap* uses an array-like structure. *TreeMap* implements the map interface using a tree-based structure.

The Java collections support generic types, although, in many cases, the developer only needs a collection of simple structures. In Java, the developer can use Java collections or arrays of objects, but both have a negative impact on performance. These are collections of objects, where the array has the pointers to objects. This representation requires extra instruction for each access and spends more space (object headers and the pointers). Moreover, in Java, it is not guaranteed that the objects are allocated in contiguous memory, thus, the spatial locality is low. Some approaches have been proposed to use the garbage collector to rearrange the objects to improve the spatial or temporal locality [Hir07]. Additionally, in Java, the primitive arrays are allocated in contiguous positions in memory. So, there is only one header for the array, and one data item can be accessed with a single instruction.

JCF does not provide containers of primitive data types, but it can use collections of objects that represent the primitive types. For this, Java provides a mechanism for converting primitive type variables into objects of the same type [HKH+16]. The solution creates an overhead [FSS13], due to primitive

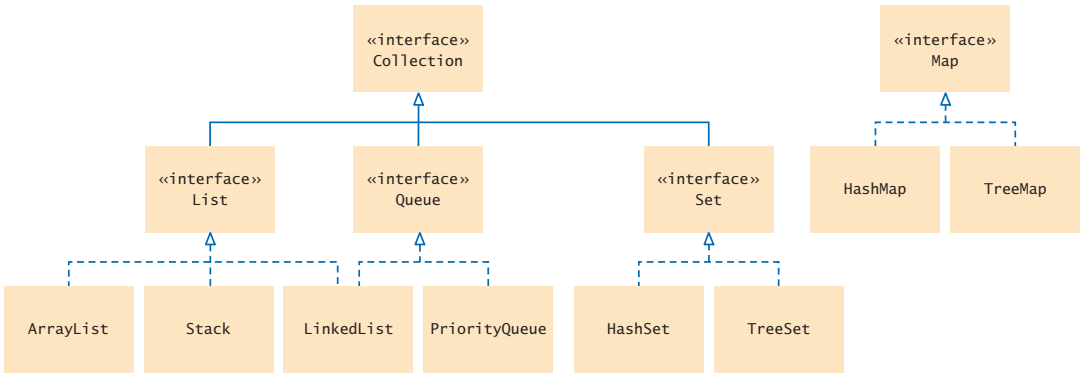


Figure 2.4: JCF class diagram [Hor16]

type conversion into an object, and the AoP layout. There are several approaches to use primitive data types arrays backed by arrays of primitive data [OW15, Tro, Vig16]. Thus, these approaches improve the performance by removing the load instructions to access the object and reducing the memory footprint (remove the object header). However, the approaches do not support structured data types and remove the domain abstractions from the code.

AoP is the best layout to support collections of heterogeneous elements (different entities). In other layouts, it is possible to implement polymorphism. However, the solution is more complex and adds overhead.

The use of arrays of primitive types for better performance can lead to abstractions removal from the code (e.g. the domain entities disappear from the code). This is common in scientific applications since the collection of objects does not obtain the best performance.

The other option encapsulates all entities in a collection and uses the primitive array for the fields (in this context, this option is called Java Collection Returns New Element(JCRNE)). So, it hides the internal collection representation to the developer creating a new entity each time the developer accesses the fields. The option keeps the entity in the code and allows the use of the SoA layout. However, it adds the overhead of creating the entities. Figure 2.5 illustrates this problem, where the entity has three fields (A, B and C), and the data fields are distributed in the three arrays. To access a field, it needs to create the new object *E* (created by *getElement*) that will be later de-constructed (by *getB* in *functionA*). *f(B)* changes the b field, but the collection is not updated. The *setB* updates the value in the new element. The *setElement* updates the element in the original collection. In this case, to update a field, it needs to copy the values multiple times (for one read and one write needs to perform seven reads and seven writes).

Java can reduce the space used by object pointers using compressed pointers. In Java, the data is typically aligned to 16 bytes, which in practice makes the last 4 bits of the address to be 0. Java takes

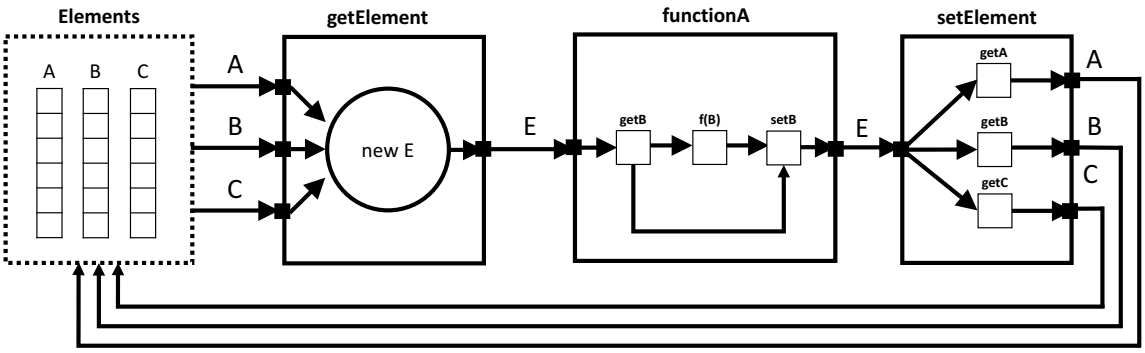


Figure 2.5: Encapsulation example

advantage of this to reduce the space used by the pointers. So in memory, it does not represent these bits, being added later when it accesses the address.

Java provides iterators to process the entire container. Iterators allow hiding the container implementation to the developer, so it is possible to process multiple containers types using the same source code. Iterators are typically a safe approach for accessing containers, as they limit access to the elements in the container. Iterators, being more abstract than using indexes, introduce more instructions. The Java dynamic compilation can remove these additional instructions in most cases.

Java iterators use the natural order to process the collection, being possible to modify the iterator. Java does not provide access to the element position by the iterator, which can prevent using iterators in some cases (e.g., transpose of the matrix). Java uses iterators to process the collection elements with *foreach* and in the streams interface.

Data sorting

The sorting technique reorders the data in memory according to the way it is accessed. This technique improves the spatial locality, but the ordering is dependent on the domain. An example in the AoP layout is the reorder of objects in memory by the collection index. If the algorithm access the consecutive position in the collection, the sorting improves the spatial locality. The technique introduces additional complexity in the program development and increases the code complexity. However, Hirzel [Hir07] made this improvement automatic by changing the Garbage Collector mechanism of a JVM. This provides the developer with a performance improvement without any programming cost.

There are complex ordering schemes, such as the Z layout however, the calculation of the data indexes can become complex leading to an overall performance degradation [TBK06].

One sorting example is present in the layout proposed by Van Emde Boas [vEBKZ76] to the Heap Sort

algorithm (figure 2.6). The work introduces a hierarchically decomposed search tree structure, implicitly decomposed by redefining index computations (for children and parent expressions) to achieve the desired access pattern. It consists of looking at the binary tree data structure from a blocked point-of-view perspective so that each memory access to the root of each block can also fetch adjacent nodes, preferably children nodes.

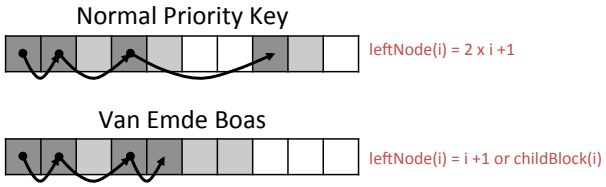


Figure 2.6: Nomal and Van Emde Boas layout (Heap access)

There are many other data sorting approaches. Overall all these approaches improve the spatial locality by making consecutive accesses in the closest memory zones.

Padding and alignment

The Padding and Alignment adjust the data footprint to specific cache/memory parameters [PNDN99] (e.g., a dimension compatible with the cache line size). The padding inserts dummy elements in the data structure to make the data structure dimension divisible or multiple at the cache line. Divisible/Multiple depends on whether the structure fits or not in the cache line. Padding is also used to remove false-sharing by inserting elements in order to avoid coherence misses.

Data alignment is essential for efficient vectorial computing in modern processing units. Java aligns the objects in memory however, in the case of arrays there is the array object header that misaligns the first element of the array. In such cases, the first array elements are processed without vectorial instructions (i.e., one by one). The problem has a more negative impact on Tiling optimisation (described next). The first tile element is misaligned, which implies a higher performance loss.

Listing 2.2 shows the code changes required to implement the padding, in this case, aligned to 16 Bytes (1 int $=>$ 4 Bytes and 1 char $=>$ 1 Byte). The structure stores three integers (12 Bytes), so it needs to increase 4 Bytes (*char pa [4]*). Nevertheless, padding does not ensure the structure alignment in memory. It is required also to align the array in memory for the cache line (see last statement pack in while on the right of listing 2.2).

Base	Padding

`//declaration`
`typdef struct s1 {`
`  int vall, va12,va13;`
`} t1;`
 	
`//allocation`
`size=sizeof(t1)*SIZE;`
 	
`//alignment array`
`ar=(*t1) malloc(size);`
	`//declaration`
	`typedef struct s1 {`
	`  int vall, va12,va13;`
	`  char pad[4];`
	`} t1`
	
	`//allocation`
	`size=sizeof(t1)*SIZE+1;`
	`ar=(struct t1*) malloc(size);`
	
	`//alignment array`
	`arl=(((int)ar+B-1)/B)*B;`

Listing 2.2: Padding - Differences of the code [LW94]

Packing

Packing creates a temporary set of data to minimise data fragmentation. The technique needs more memory since it replicates some of the data but reduces the capacity misses. Thus, its use makes the size of data that is loaded into the cache smaller. The optimisation is typically combined with the tiling optimisation, performing packing while loading a tile. Packing optimisation improves spatial locality.

The optimisation can be applied in two ways: first, creates all packed structures at the start; second, creates only a packing structure and loads the data to the packing structure when needed. In the first case, there is higher memory consumption. However, the packing cost is all at the beginning. The second option reduces memory consumption, but it may force to load several times the same data.

The packing combined with parallel execution can reduce coherence misses since each processor can use its data pack. Furthermore, it can put the data closer to the processing unit (in its main memory in a NUMA system). If each thread initialises its data pack, the memory management system will allocate that memory in a place closer to the processing unit.

The listing 2.3 shows the packing in a matrix multiplication for part of the matrix B. Before the calculation, it needs to load data from matrix B (*B*) into the packing matrix (*bb*). As matrix B is not rewritten in this calculation, the matrix update is not performed at the end.

2.2.2 Data flow

The first section presented optimisations that modify the data footprint in memory. This section presents optimisations that modify the order of processing the elements.

```
for (int jj = 0; jj < size; jj += tilej) {

  //Packing matrix B
  for (int k = 0; k < size; k++)
    for (int j = 0; j < tilej; j++)
      bb[k][j] = B[k][jj + j];
  (...)
}
```

Listing 2.3: Packing implementation - copy data to packing matrix

Tiling

The general approach is a divide and conquer strategy: the problem is decomposed into subproblems of smaller size that fit into faster memory [YRP+07]. This optimisation intends to take advantage of the temporal locality present in the algorithm since each subproblem is in one of the cache levels. However, it increases the number of instructions executed due to two factors:

- more loop control instructions since it introduces more and smaller loops.
- more load and store instructions since it performs additional steps through the data.

The tiling optimisation needs an additional parameter that defines the smaller problems size (number of blocks). The size defined for the smaller problems has an impact on the number of instructions, and cache misses. If the size is small the number of instructions will be higher, but if the parameter is too large it cannot take advantage of the faster cache levels due to capacity misses. This technique can be applied multiple times, thus enabling a better fitting of the problem to multiple cache levels.

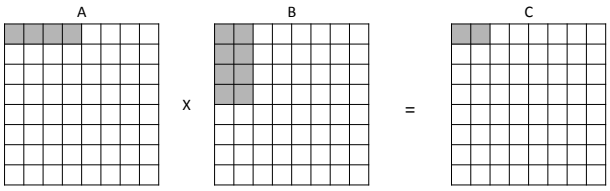


Figure 2.7: Loop tiling - data needed to calculate the first elements (8 iterations) on matrix multiplication.

Figure 2.7 illustrates a matrix multiplication algorithm where the data is accessed by blocks, thereby decomposing the problem into subproblems. With this decomposition, it is intended that subproblems have a size smaller than the level of the cache that it is intended to optimise for.

Tiling optimisation is probably the most used optimisation since it can provide huge improvements in certain algorithms, namely in those that can take strong advantage of temporal locality. Let us consider the following example: collections *p1* and *p2* have both 1024 elements and each occupies 24MiB in memory;

the numbers of tiles is 4; the cache size is 12MiB; the computation performs two loops picking one of *p1* and performing a computation with each *p2* element; only one element of each collection is required for each computation; the elements of the collection do not exhibit spatial locality (simplifies the example). In a simple implementation, *p1* collection elements will be read from memory only once. However, the collection *p2* will be read from memory in all iterations of the outer loop (1024), reading $\sim$ 24GiB from memory. After applying the tiling optimisation, the original function is called 4 times, which implies that the collection *p1* is loaded 4 times from memory. Each call is performed over 4 subcollections of the original *p2* collection that occupy only 6MiB. This subcollection fits in cache, which implies that during the processing of the inner loop the data is read from the cache and not from to main memory. There is a substantial reduction in the number of data read from memory, in this case only 120MiB is required.

In the example, more loop control instructions are added, as well as data reading instructions. The data from the p1 collection is read-only once, with tiling optimisation of the data.

Loop fusion

The loop fusion merges multiple loops over a data set into the same loop. Therefore the instructions and the data accesses reduce (fewer instructions for controlling the loops), which implies a decrease in the number of misses. However, the technique increases the code inside in the same loop body, so, it can increase the number of misses in the instruction cache.

Loop	Loop fusion

`for i = 1 to N`
`  M[i] += C1`
 	
`for i = 1 to N`
`  M[i] *= C2`
	`for i = 1 to N`
	`  M[i] += C1`
	`  M[i] *= C2`

Listing 2.4: Loop fusion - Differences of the code

This technique can be applied when several functions are applied to the same data (listing 2.4). In this example, it applies two operations to the same data. Thus, it needs to traverse the data twice, but the optimisation applies the two operations on a single loop. In the example, the optimisation reduces the memory accesses, which implies a decrease in the load/store instructions, and in the cache misses. Additionally, it allows taking advantage of the multiple functional units into the processors, reducing the Cycles per Instruction (CPI).

Loop reorder

Loop Reorder changes the order in which data is processed. As a rule, the optimisation aims to improve the data locality (e.g., accessing the matrices by rows instead of columns). This reordering of loops is used, for example, in matrix multiplication [SG98], obtaining a significant impact on performance. Typically, changing the order of the loops is simple for the developer. However, validating the optimisation can be difficult.

In the example of listing 2.5, assuming an array of doubles (8 bytes) and a cache line of 64 bytes, for each load made from the main memory 64 bytes are loaded, and only 8 bytes are used. In the second case, as the array is accessed by rows and the array elements are in consecutive memory, the 64 bytes loaded into the cache are fully used.

Access by column	Access by line

`for j = 1 to N`
`  for i = 1 to N`
`  M[i][j] *= C2`
	`for i = 1 to N`
	`  for j = 1 to N`
	`  M[i][j] *= C2`

Listing 2.5: Loop reorder - Differences of the code

2.2.3 Summary

The table 2.2 summarises the locality optimisation impact on a program. The first column (Miss type) shows the type of the miss that each optimisation intends to solve (cold, capacity, cold, conflict or coherence). The Locality column refers to the locality type that optimisation improves. Finally, the last column shows which modifications to the code are necessary at the implementation and in which part of the program the optimisation improves.

	Miss type	Locality	Implementation -> Impact
Data Layout(AoS->SoA)	Capacity	Spatial	All -> All
Sorting data	Cold	Spatial	Local++ -> Local+
Padding and alignment	Conflicts+Coherence	Spatial	Local -> All
Packing	Capacity	Spatial	Local -> Local
Tiling	Capacity	Temporal	Local -> Local
Loop fusion	Capacity	Temporal	Local ->Local
Loop reorder	Cold+Capacity	Spatial+Temporal	Local -> Local

Table 2.2: Impact the memory optimisations in the program

A change from AoS to SoA reduces unnecessary data cache loads so, it minimises capacity misses. The optimisation has a massive impact on code since the developer needs to change all data accesses.

In practice, this optimisation avoids loading additional fields from the structure into the cache (reduces the data loaded from memory). Sorting data reduces cold misses. This optimisation reorders the data in a specific program step to improve spatial locality. Padding and Alignment are specified in the data structures definition step, so its impact on the development cost is minimal. This optimisation reduces conflicts and coherence misses. Packing aims to reduce capacity misses by reducing the data loaded into the cache that is not needed. The Packing has a performance impact only where the optimisation is applied. Tiling decomposes the problem to reduce capacity misses. This optimisation is applied to a specific function/loop. Loop Fusion takes advantage of the data loading to cache to perform more calculations over these data. Thus, it reduces capacity misses. Loop Reorder improves the spatial and temporal locality (such as processing the matrix by lines reduces misses). On the other hand, Loop Reorder allows maintained the problem segment at the cache, reducing misses due to capacity.

In short, the layout change is the optimisation that implies an intensive transformation in the domain code. Tiling and the usage of the best layouts for performance make code less legible.

2.3 Java

In this section, some relevant characteristics of Java are described. Java compiles the code to bytecode that executes on several platforms where a Java Virtual Machine(JVM) is available. The JVM transforms the bytecode into native instructions of the machine running the program [Oak14]. This process will be described in the following subsection. The subsequent Garbage Collector section describes how memory is managed in Java. The final section describes the parallelism model in Java.

2.3.1 Compiler and execution processes

A Java program (figure 2.8) is first transformed into bytecode (generated by the Java Compiler), code that is platform-independent. In this step, syntax errors are detected, and some optimisations are applied (e.g., *final* variables are replaced by the value). In this step, the applied optimisations are platform-independent. The bytecode does not execute directly in the machine, it needs a JVM to run upon. Therefore, this section will explain the execution process in the JVM [Oak14].

In the execution step, the JVM interprets or compiles the bytecode. Interpretation occurs when the code is not yet compiled. Thus, the execution starts immediately without waiting for the compilation of code. However, the interpretation mode is slow. For this reason, the JVM compiles the heaviest methods. Just-in-Time(JiT) uses dynamic compilation which compiles the code during the execution step. The

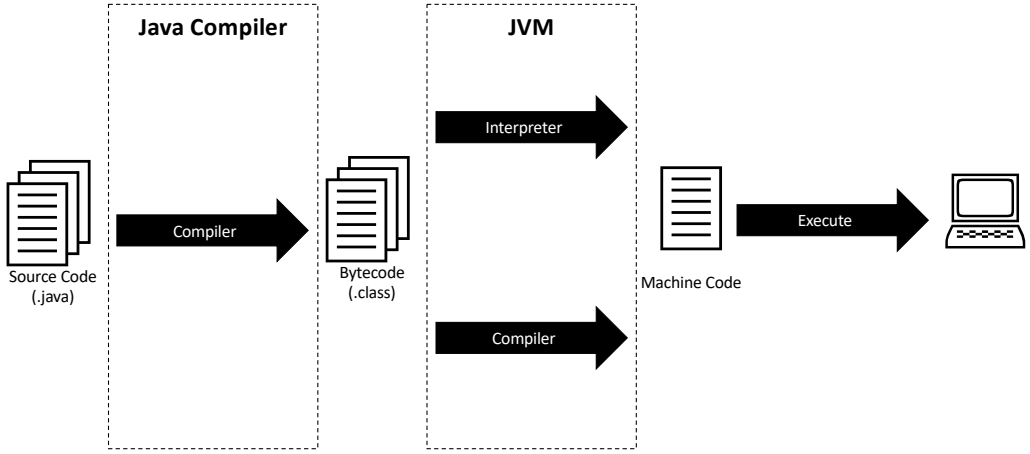


Figure 2.8: Java execution process

dynamic compilation allows the application of more aggressive optimisation techniques (e.g., dynamic compilation allows de-virtualisation of the methods [SOK+04, APC+96]). This compilation process increases the efficiency of Java when compared with techniques that use static compilation.

Compiling the code occurs asynchronously: the program is not stopped to compile the code. When the method/code is selected for compilation, it is placed in a queue. After the compilation of the method/code ends, that part of the code runs natively in the machine language.

The compilation process enables several optimisations of the code: In-line expansion, Escape Analysis, De-optimisation and others. In-line expansion inserts the method code in the location where the method is invoked, removing the invocation cost. The optimisation is essential to remove the data encapsulation cost when the objects fields are accessed by *get/set*. These small methods can introduce an overhead, but compiler optimisation can avoid it. The methods are expanded in two distinct situations [Oak14]: when the method size is less than a value (by default 35 bytes) or when the method is invoked N times and is not extensive (by default 325 bytes).

Escape analysis is one of the most sophisticated optimisations in the JVM. It allows the compiler to improve the objects allocation performance, and to remove the overhead of synchronisation [CGS+99]. It analyses the accesses to objects, to identify the methods or threads that access the objects. The analysis allows deciding if the object is created or not (e.g., allocate space for the object fields in the stack instead of creating a new object). The escape analysis allows knowing if the object is accessed by a single thread, in this case, it removes the synchronisation operations.

Sometimes the compiler enables optimisations that are valid at that compile-time and become invalid later. When the optimisation becomes invalid, the JVM runs a version less optimised (this version is

available in a stack of compiled methods/code). An example is the methods called through an interface that is expanded in-line, and later, it is called with a different concrete class method. The de-optimisation of a code does not permanently disable optimisations. The code de-optimisation allows more aggressive optimisations for a specific case, which translates into better overall performance.

In more abstract languages, like Java, this set of optimisations is essential for performance. They allow removing the overhead created by the language implementation due to the abstraction support and are fundamental to make the proposed work feasible in terms of performance. To summarise, the dynamic compilation enables many opportunities to improve performance: applying optimisations that are valid for the specific data input; and optimised to available resources. However, runtime compilation uses processor time, and this may interfere with the runtime of the application.

Other relevant optimisations

The loop unrolling provides the calculation of multiple elements in the same loop body. Typically, the JiT compiler implements the loop unrolling optimisation although, it can also be applied manually by the developer. The listing 2.6 shows the assembly codes with and without the optimisation. The listing shows that there is a significant reduction in the number of instructions executed. In the code without optimisation, for each element, four instructions are needed. With loop unrolling, it needs seven instructions to calculate four elements (1.75 instructions/element).

```
.L6:                                    .L6:
  addl  %ebx, (%edx,%eax,4)               addl  %ecx, (%edx,%eax,4)
                                          addl  %ecx, 4(%edx,%eax,4)
                                          addl  %ecx, 8(%edx,%eax,4)
                                          addl  %ecx, 12(%edx,%eax,4)
  incl  %eax                              addl  $4, %eax
  cmpl  %ecx, %eax                        cmpl  %ebx, %eax
  jl  .L6                                 jl  .L6
```

Listing 2.6: Loop unrolling optimisation

The optimisation reduces the number of instructions executed: less loop control instructions since each loop body processes more elements; improves the instructions scheduling [Int16]; simplifies the data mapping into registers; allows to apply of other optimisations (removal of redundant loads and subexpressions). However, its use increases the code size, which can increase the number of misses. Moreover, it can inhibit the use of branch prediction. That happens when the loop body has many jump expressions, leaving the branch predictor without storage capacity.

Current Java compilers generate vectorial instructions that allow the calculation of several elements in a single instruction [NCL+10]. The vectorial instructions follow the SIMD approach, where the same instruction can process several elements. The compilers generate vectorial instructions when possible. In some cases, the developer must modify the code to enable this auto-vectorisation. The fundamental condition is that the same operation is carried out on several elements. Moreover, the elements must be in consecutive memory positions, making the SoA layout the most efficient for vectorisation [JRS16]. The use of the AoS layout creates additional overhead, as it requires scatter and gather operations.

The vectorisation is more efficient when elements are aligned in memory (the first element must be aligned in memory, normally at a 32 Byte boundary). The JVM supports auto-vectorising. However, there is an additional problem since the first array element is not aligned in memory.

The vectorial instructions can reduce the processor frequency. These instructions typically have a higher power consumption which causes a decrease in the processor frequency [Len14].

2.3.2 Memory management

In Java, the objects are allocated in the heap, while primitive data and (local) object pointers are in the stack. The memory allocation and freeing in the stack, as in other languages, is trivial: when the function ends, the space used for local data is released by updating the stack pointer.

All objects data is kept in the heap. The developer creates the objects explicitly, but in Java, the object destruction is the responsibility of the Garbage Collector(GC). The GC performs several functions: find unused objects, make space available in memory and remove empty spaces between data. When finding unused objects, and if more memory is needed, the GC frees up that space. Removing empty spaces allows better management and improves access efficiency, which is why the GC must keep data spaces in compact regions.

The GC is activated when the free heap space is limited (space is need for other data). For this reason, the larger heaps size implies that the mechanism is activated less often. However, using a large heap means that each call to manage the heap has a higher cost (more data is analysed). The JVM[2] does not recommend the use of heap sizes larger than the main memory.

When the GC is activated, the heap content is analysed. The typical GC uses two spaces: the young generation and the old generation. Allocated objects are placed provisionally in the young generation. If the objects remain active, they are placed in the old generation. This division allows the GC to be more efficient in memory management. When the young generation fills up, objects that are no longer used are

[2] https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/memman.html

discarded (e.g., auxiliary objects that exist only in the method context that has finished). The objects more persistent (e.g., objects created in the main method) are moved to the old generation. After this process, the young generation becomes empty. Using this mechanism allows the GC to minimise the impact of creating many objects. Moreover, this process reduces data fragmentation since temporary data no longer cohabits with more persistent data.

The JVM uses different algorithms for the GC: Serial GC, Throughput, CMS and G1. Serial GC uses a single thread to run the GC and interrupts all application threads. Throughput is used by default in the 64-bit JVM. This GC uses several threads but also interrupts all application threads. The CMS is designed to avoid long pauses. It uses several threads in the background when there is processing time available if not, it is executed in a single thread. The application threads are interrupted for short periods. G1 collector is designed for heaps greater than 4 GiB. For this purpose, it subdivides the old generation into several regions.

2.3.3 Parallelism

Java supports thread-based shared memory programming natively. In Java, parallelism can be used for multiple objectives. In this thesis, the objective is to improve the performance of the scientific application. Scientific applications require massive processing, and parallelism allows us to take advantage of multicore machines. In scientific applications, many problems can be divided into multiple subproblems that can be processed by multiple threads. Typically the number of subproblems should be close to the number of cores. The use of an excessive number of threads causes more context switching, delaying processing. On the other hand, the insufficient number of threads does not allow to take advantage of the full capacities of the processor [Sub11].

In Java, there is a set of mechanisms for parallel processing. Java has the basic thread constructors in which the threads are managed manually. On the other hand, Java supports executors, where the developer creates tasks and submits them for execution. In programming code, these two options are similar, but the latter avoids the explicit management of the lifetime of threads. Java 8 introduced a new distinct concept, streams. The streams provide several higher-order functions that can be performed in parallel. The stream concept uses an API closer to the domain, but the use of streams is limited. Streams were developed to process data in collections in the context of the JCF.

Chapter 3

Proposed approach

The main goal of the approach is to promote high-level programming in data-parallel scientific applications. Typically, scientific applications need to leverage all computer power available by using High-Performance Computing (HPC) techniques. In HPC, the main focus is to reduce the program execution time. Programming using domain abstractions simplifies the development and makes the code more understandable but can reduce performance. In these cases, the developer removes the abstractions from the code to increase performance. The abstractions represent entities in the domain and actions on them. Thus, the approach should enable domain abstractions while keeping high performance.

The approach should provide a framework for the developer with the following capabilities:

1. use domain abstractions in the code, using an OOP approach;
2. deliver high performance;
3. allow data locality optimisations in the final development step;
4. support common data locality optimisations, e.g. data layout and tiling;
5. support data-parallel processing;
6. do not constrain the most important compiler optimisations.

The OOP enables writing code using abstractions from the domain that simplifies the code and writing of complex software. These languages use encapsulation to hide the internal representation of objects and leave only the relevant interface exposed to the developer. Traditionally, the optimisation techniques modify the code and tune it for a specific execution platform, making code efficiency dependent on that

platform. Frequently, the domain abstractions are removed, and the encapsulation is lost. The proposed approach should enable the object oriented concepts with a performance compatible with the one required by HPC problems. In this case, the domain code uses the Java language to enable the object oriented concepts. The approach must support the main OOP concepts (objects, classes, and inheritance [Weg87]). Additionally, the approach has to maintain compatibility with Java.

The approach should allow the developer to build the code in two distinct steps: a domain code development step and an optimisation step. In the first step, the developer must specify the code with domain concepts. At the end of this step, the developer should have a code that satisfies the functional requirements. In the second step, the developer has at his disposal a set of optimisations for performance tuning. The proposed approach aims to avoid premature optimisations in the domain code and support tuning to a specific platform in a simple way. The optimisation step must maintain the domain concepts created in the first step.

One essential program optimisation in scientific applications is the data layout. The best data layout depends on the execution platform. The approach must support multiple data layouts to be selected in the final development step (i.e., platform-specific tuning). Developers select from one of the layouts available or create a new one. However, changing the layout should not have an impact on the domain code.

Overall, the approach aims to improve performance by supporting multiple data layouts and using other data-related optimisations. The main focus is the collections accesses performance. For this purpose, the developer can change the collection layout or change the order of accessing data elements. Thus, the approach has to provide efficient layouts for collections and offer abstractions to implement the most common techniques in HPC, such as tiling and packing.

Another requirement is the simplification of parallel programming by supporting parallelism patterns over collections. The developer defines the operations to perform in parallel, and if required, decides the strategy to deal with concurrent data accesses. For this, the developer must use Java mechanisms or thread private data. The tool should support thread private data without requiring modifications to the domain code.

Finally, the solution should not constrain the most common compiler optimisations. Some examples are auto-vectorisation and loop unrolling. The approach uses abstract data layouts at the programming level (1st requirement), but these layouts are incompatible with vectorial instructions (e.g. collections of objects). However, the approach could internally use SoA layout to enable vectorisation in more cases.

The next section presents an approach overview and how it satisfies these requirements. The *Programming interface* section shows how developers can use the framework, including the mechanism to apply optimisations. The following section presents an overview of the implementation. Finally, the last

section presents the most relevant limitations of the current tools.

3.1 Overview

The approach includes a methodology and a toolset that, as a whole, support the development of abstract code that can be tuned to obtain high performance.

Traditionally, in scientific applications, the developer accesses the data directly (use the entity position in arrays of raw data). In the proposed approach, the developer privileges more abstract access methods: iterators and higher-order functions. Java iterators hide the implementation of the collection. However, to support efficient data layouts, the approach needs to hide both the collection and entities implementations. The approach supports the Java collections API but requires the usage of getter and setter methods for accessing the object data fields. This combination allows hiding the layout representation and providing efficient data access, namely by using a SoA collection implementation.

The approach supports two generic types of locality optimisations (2nd requirement):

1. Layout change —the strategy provides multiples layouts for collections (e.g., AoP and SoA) with the same access API. The developer creates the domain code without depending on the internal data representation. The SoA layout implementation provides greater efficiency without removing the abstractions from the code (1st requirement).

2. Change the execution flow —the approach provides a high-level constructor that allows applying optimisations that change the execution flow in the final development step. Thus, the optimisation step is independent of the domain code development (3rd requirement).

The developer specifies the domain code without being concerned with optimisations (figure 3.1). For this, the approach provides a high-level API compatible with the Java collections API. The API provides collections, iterators and higher-order functions for processing collections of objects. In a second step, the developer specifies the layout for those collections and other optimisations. Those optimisations are specified by annotations that make the optimisation code pluggable (it is possible to enable or disable those optimisations).

The *Domain* specification has three steps (blue boxes in figure 3.1). First, the developer designs the domain model in the Unified Modelling Language (UML). The developer represents the domain concepts and the relationship between them. Second, the approach uses the domain model to generate a library containing collections implementations. The compositions relationships of 1 to N are converted into the corresponding collections. Third, the developer writes the domain code using Java interfaces without

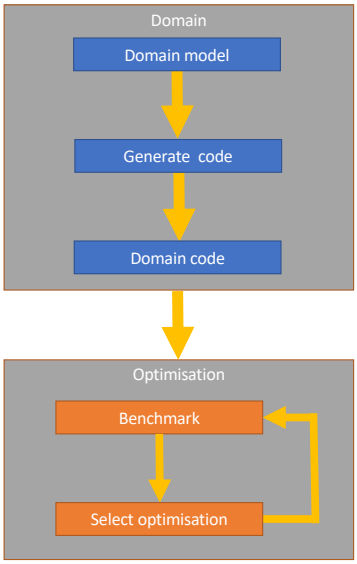


Figure 3.1: Approach workflow

knowing the data layout implementation. For this step, the framework provides three levels of abstraction. At the first (lowest) level is the indexed access, using the collection *get* and *set* methods. The second level uses iterators, which remove the loop index from the domain code. The highest-level uses higher-order functions which removes the loop in the code.

At the end of the *Domain* step, the program already implements all the functional requirements. The program performance is not relevant at this first step: the emphasis is on program abstraction and correctness. In the *Optimisation* step, the performance is analysed and improved. The developer analyses the program execution and identifies the code parts that should be optimised using profiling tools (*Benchmark* step). The approach includes a tool that enables access to the processor counters in Java to help the developer on this step. In the *Select optimisation*, the developer can optimise the program using two distinct mechanisms: layout and domain decomposition (*gSplitMapJoin* mechanism). In this step, the developer can select a data layout through an application parameter.

The domain decomposition allows the developer to apply multiple optimisations, such as tiling, packing and parallel execution. The mechanism improves the abstraction level since the developer defines how to decomposition the domain, and the approach hides implementation details. The mechanism allows dividing collections to define the subdomains. The mechanism starts by decomposing the domain, processes all subdomains and finally aggregates the results. The traditional tiling implementation (section 2.2.2) injects a new code into the program without any meaning for the domain (e.g., a new loop). The proposed approach increases the abstraction level since one annotation specifies all the optimisation

parameters.

The approach was designed to support the two most common locality optimisations: layout change and tiling. The strategy for changing the layout relies on two software design patterns: *Proxy* and *Iterator*. The idea is to use an interface to support the domain API (figure 3.2) where data entities are accessed through an intermediate *Proxy*, and that is also an *Iterator* to iterate over the collection. Thus, the *Proxy* has the same methods as the domain entity and provides methods to iterate over the collection, hiding the concrete collection layout and the entity implementations. The other strategy allows the developer to decompose the problem domain into subdomains. The key is to manipulate the parameters of the method that processes the domain. The developer inserts an annotation where it specifies how to divide the collections. The tool generates a new method that decomposes the domain and processes each subdomain. Besides tiling, this strategy enables several other optimisations, such as packing and parallel execution. Listing 3.1 exemplifies how the tiling optimisation is specified. The example will execute the original code (2 internal loops) several times each time with a subcollection of the p2 *gCollection* parameter (in practice, a new external loop is added).

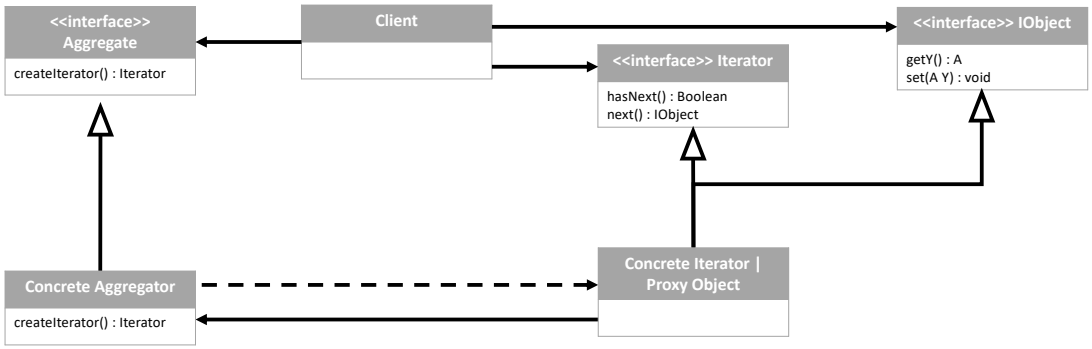


Figure 3.2: Iterator and Proxy pattern [GHJV93, Met02]

```
@gSplitMapJoin(name = "Tiling", map = "Sequential", split = {none ,"Virtual"}, reduce = {"
    default", "default"})
f(gCollection p1, gCollection p2){
  for(gIterator it1=p1.begin(); it1.isless(p1.end()); it1.inc()){
    for(gIterator it2=p2.begin(); it2.isless(p2.end()); it2.inc()){
      (...)
    }
  }
}
```

Listing 3.1: Tiling specification by *gSplitMapJoin*

These two strategies allow the developer to apply common optimisations (4th requirement). The second strategy also satisfies the 5th requirement by supporting the processing of the subdomains in

parallel. Additionally, the developer can use the thread private data mechanism. In this case, the developer defines which fields are private, and the tool generates a new structure to implement this mechanism.

All mechanisms were developed to be compatible with important compiler optimisations. In the first case, using a proxy/iterator allows the compiler to remove the additional code through method in-lining and escape analysis (to avoid proxy/iterator object creation) enabling further compiler optimisations. The second strategy has an additional cost but does not inhibit the compiler optimisations.

footnote

3.2 Programming interface

This section shows how to use the approach presenting a concrete example (MD case study from JGF [BSW+00]). First, it shows how to write the domain code. Then, it explains how to apply each optimisation.

3.2.1 Domain specification

Domain model and code generation

The first step is to create the domain model (figure 3.3 shows the example. The approach provides a tool to specify the domain model and to generate code to support the model.

The process starts with a UML class diagram, where the developer specifies the domain entities. The developer specifies the entities that compose the domain and attributes. The tool generates collections to support the different collections layouts for each entity from the model. The entities specified should have a composition relationship from 1 to N[1]. The current tool does not allow polymorphic entities (collection entities must belong to the same class).

The generated code is organised into a set of packages that contains the interfaces and classes to support the model. For the MD case study, the tool generates two packages: *ParticleCollection* and *gCollection*.

ParticleCollection package (figure 3.3) contains all the interfaces. These interfaces are domain-dependent. This package provides the API that allows accessing the domain entities. For MD case study, the tool generates two interfaces: *Particle* and *gVector*. The *Particle* interface represents a particle and has the methods to access the particle attributes.

[1] In the tool prototype, aggregation relationships are not supported.

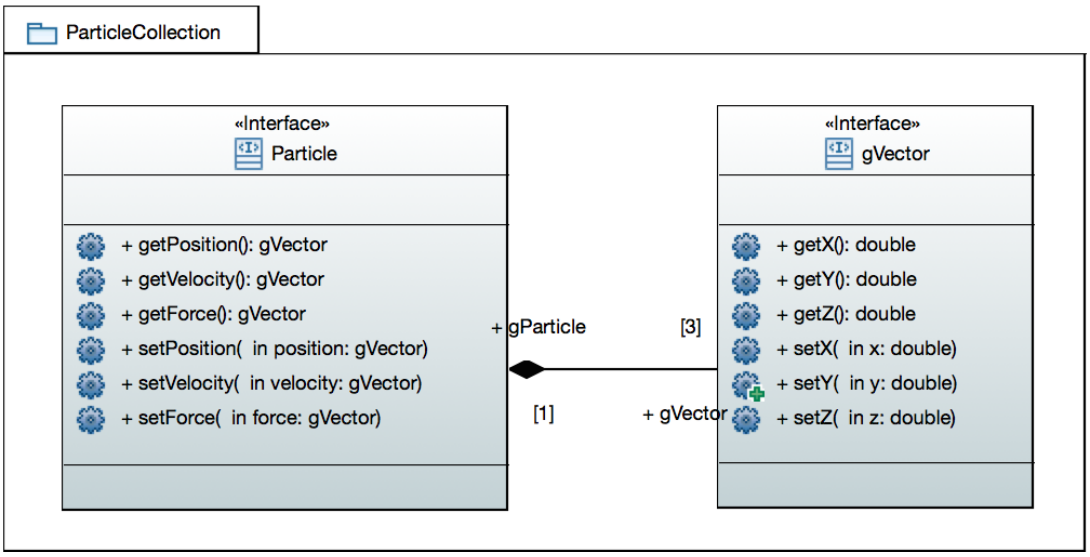


Figure 3.3: *Particle* interface (UML tool)

This package also contains additional classes to help the developer to create collections. The first class allows the developer to create a single *Particle* (class *realParticle*). The second creates a collection of *Particles* (class *FactorygCollectionParticle*). There are similar classes for the *gVector*.

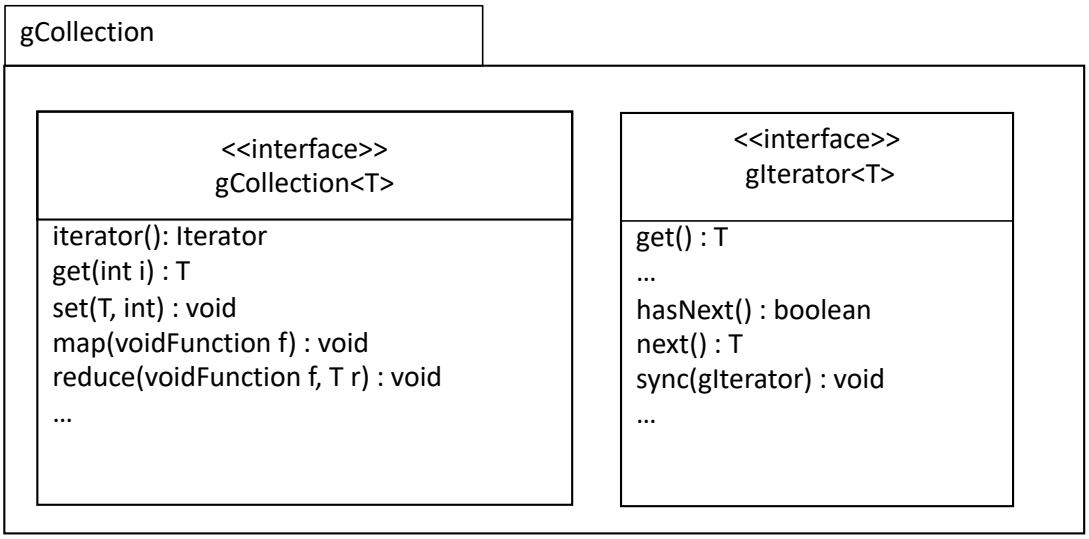


Figure 3.4: The most important interfaces to access the *gCollection*

The *gCollection* package (figure 3.4) has the interfaces to support the collection. This package is the same for all domains, but it is also generated by the tool. The most relevant interfaces in this package are

the *gCollection* and *gIterator*.

All collections implement the *gCollection* interface. The interface allows index access to each element (by the get and set), creation of iterators (by *iterator*, *begin*, *end*), and the application of a method to all elements of a collection (*reduce* and *map*). *gCollection* interface extends the Java *List* interface which makes it compatible with the Java API. The *gCollection* interface also provides a set of methods that allow the developer to divide the collection into subcollections and sorting the data to improve the spatial locality.

Domain code

In this step, the developer writes the domain code using the collections API. To create a collection, the developer uses the factory pattern [MW06, Gra02]. In the listing 3.2, the developer creates a collection of particles. For this, the developer creates a factory of collections of *Particle*. Subsequently, the developer uses the factory to create a collection with the specified size.

```
FactorygCollectionParticle factory = new FactorygCollectionParticle();
gCollection<Particle> mdCollection = factory.creategCollection( size );
```

Listing 3.2: Example of the creation of a *gCollection* of particles

The framework allows the entity to exist outside the collection. The developer can insert the entity into the collection by copy. The listing 3.3 shows how the developer creates a particle. Initially, the developer creates three *gVector*. Later, it creates a particle using these three *gVector*.

```
gVector position    = (gVector)  new realgVector( px, py, pz);
gVector velocity    = (gVector)  new realgVector( vx, vy, vz);
gVector force       = (gVector)  new realgVector( fx, fy, fz);
Particle particle   = (Particle) new realParticle( position, velocity, force);
```

Listing 3.3: Example of a *Particle* creation

To add the *Particle* to the collection, the developer uses the *add* (listing 3.4). At this step, the framework API is slightly different from Java. Java adds the object to the collection, but the proposed framework copies the object values to the collection. There is also the *set* that specifies the position that the element will occupy in the collection.

```
mdCollection.add(particle);
```

Listing 3.4: Example of adding a particle to a collection

For accessing collections, the approach provides several methods using different abstraction levels (listing 3.5): index, iterators and higher-order functions. In the index access, the developer specifies the index using the *get(i)*. In the example, the developer obtained the particle *aP1* through this access type (this is a more traditional way of collections accesses). Index access is flexible to iterate over the collection since the developer has full control over the index range. On the other hand, index access exposes the processing order in the domain code, restricting the set of optimisations that can be applied later.

```
//index access
for(int i=0; i<size; i++){
  Particle aP1 = mdCollection.get(i);
  System.out.println(aP1);
}

//iterator access
Iterator<Particle> it = mdCollection.iterator();
while(it.hasNext()){
  Particle aP2 = it.next();
  System.out.println(aP2);
}

//higher-order functions
mdCollection.stream().forEach(aP3 -> System.out.println(aP3));
```

Listing 3.5: Options to access collections

The second case uses iterators for the particle *aP2* (i.e., Java iterators). The approach also provides *STL iterators* [SL95]. Thus, the developer has available a *begin()* that obtains an iterator for the collection begin. To test if there are more elements to process, the developer compares the iterator with a new iterator that is obtained by *end()*. For comparison, it uses the *isless(...)*. To advance to the next element in the collection, the developer uses *inc()*. Moreover, the *glterator* allows forward and backward iteration by *inc* and *dec* (like the random iterator available in STL).

Finally, the approach supports higher-order functions. In the example, *aP3* is obtained with this type of access. Additionally, the approach provides two other higher-order functions (listing 3.6)[2]. The *map* processes each collection element by applying the provided method. The *reduce* reduces the collection to a single element by applying a reduce operation. The second parameter of *reduce* stores the result and should be initialised with the neutral element.

Java iterators and streams do not support iterating several collections at the same time. The approach provides a *sync* for this purpose. The *sync* updates the iterator position with the same value as another iterator.

Listing 3.7 illustrates the program to compute the force between a *Particle p1* and the other particles

[2] *map* and *reduce* are an alternative to *Java streams*: *for-each* and *reduce* respectively.

```
...
collection.map( it -> it.setgDouble(it.getgDouble() + 1 ) );
...
gDouble returnvalue = new realgDouble(0.0);
collection.reduce((it, res) -> res.setgDouble(it.getgDouble()+res.getgDouble()), returnvalue);
...
```

Listing 3.6: Higher-order functions examples

in a collection. To obtain the *Position* vector, the developer uses *getPosition*, and then, accesses to the vector fields through the *getX*, *getY* and *getZ*. To access to *particleSet*, the developer uses a Java iterator and a *while* loop to process all elements. Note that *Particle*, *gCollection* and *gIterator* are interfaces, that will be implemented by concrete classes according to the layout (this code is layout independent).

```
// the same method for all data layouts
void forceParticle(Particle p1, gCollection<Particle> particleSet) {

   // get coordinates of particle p1
    xi = p1.getPosition().getX();
    yi = p1.getPosition().getY();
    zi = p1.getPosition().getZ();

    // iterate over particleSet
    Iterator<Particle> iterator = particleSet.iterator();
    while(iterator.hasNext()){
       Particle p2 = iterator.next();

       // compute distance
       xx = xi - p2.getPosition().getX();
       yy = yi - p2.getPosition().getY();
       zz = zi - p2.getPosition().getZ();
       (...)
    }
}
```

Listing 3.7: Example of the usage of the data API

3.2.2 Optimisation specification

Data Layout

The approach makes it possible to develop a program where the code has the domain entities (e.g., Particle interface in the example of listing 3.7), but the layout details are hidden (the developer writes the code using an API closer to the AoP layout). The layout can be selected subsequently in the execution step, making it possible to test layouts easily. The developed tool generates two layouts by default: AoP and

SoA. The developer uses the *Factory method pattern* to choose the layout (see listing 3.2). The developer can provide one additional parameter to the *creategCollection* with the layout[3].

It is also possible to change the layout on a specific part of the program. For this, the developer uses the packing optimisation that will be presented later.

Domain decomposition

The domain decomposition enables several optimisations. The developed approach changes the method parameters by dividing the collections, thereby the problem is processed as several subproblems. For each *gCollection*, the developer defines the option for domain decomposition. On the other hand, there are several options to aggregate the subcollection after the processing. The developer inserts one annotation in the original method (listing 3.1), and the tool generates a new method. The new method calls the original method with each subdomain. The developer can choose to call the original or optimised method. Thus, the optimisation is pluggable (the developer selects between the original or optimised method to execute).

In the annotation, the developer must specify the name, the map to use (sequential or parallel), the *split*s and *join*s. In practice, listing 3.1 applies the tiling optimisation to the *f* method. The annotation processing tool creates a new method (*Tiling.f(gCollection p1, gCollection p2)*) that calls the original method n times (each time with one of the subdomains). In this case, it decomposes the collection *p2* into several collections. The developer specified the *Virtual* split since it has a lower cost. The other alternatives need to replicate data that imply more memory required and data copies.

The tool selects the number of partitions to generate (the default value is 2), but it can be defined by a virtual machine argument (*Dtile* option). The section 3.3 contains details about how the new class is generated.

Current processors have multiple cache levels, so, sometimes, it is more efficient to partition the collection several times. The approach supports tiling with multiple levels since the same mechanism can be applied again by annotating the generated method (*Tiling.f(...)*).

Packing

The annotation tool supports packing optimisation. In listing 3.1, the collection *p2* is decomposed using the *Virtual* split. The method creates views over the original *P2* collection. Alternatively, the developer can

[3]Can be an application parameter; *aop* for layout AoP and *soa* for SoA layout.

choose packing to divide the collection where the data of each subdomain is copied into a new collection. There are two packing alternatives: *Packing* and *PackingOnDemand*.

Packing creates all subcollections in the initial step and writes the results, into the original collection in the final step.

PackingOnDemand creates a subcollection and copies the data into the subcollection only when necessary. After processing the subproblem, the data is rewritten into the original collection. If the processing is performed in parallel, there is one subcollection for each thread.

The approach also allows changing the layout of a collection or subcollection. For this purpose, the developer can use *PackingSoA* or *PackingOnDemandSoA* in the split specification (listing 3.8).

```
@gSplitMapJoin(name="f1", map="map", split={"none", "PackingSoA"}, join={"none", "default"})
```

Listing 3.8: Example of a change in the layout in the packing of a collection.

Parallelism

The current approach implementation has several options to implement *map*s and *reduce*s including sequential and parallel versions. The sequential version processes the elements[4] by the natural order. The developer has several schedulers available for *map* and *reduces* with parallel processing: *ParallelBlock*, *ParallelBlockBalance*, *ParallelCycle* and *ParallelDynamic*. Figure 3.5 illustrates the supported scheduling of a collection with twelve elements and three threads.

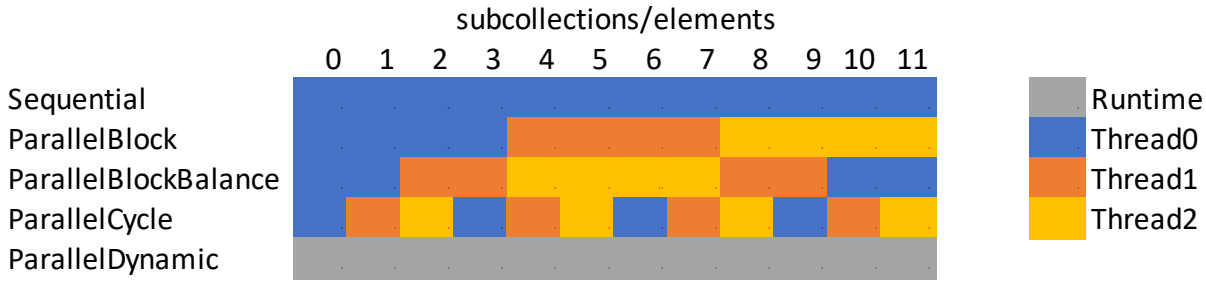


Figure 3.5: Map task onto threads (collection with twelve elements and three threads)

The *ParallelBlock* aggregates elements into blocks. The block size is the number of elements divided by the number of threads. If the number of elements is not divisible by the number of threads, the last block is larger.

[4] An element can be an entity or subcollection.

ParallelBlockBalance improves the performance in the cases where the computation reduces or increases monotonously among collection elements. *ParallelBlockBalance* distributes the block X to the thread with the same number in the first half of processing. In the second half, the blocks are processed in reverse order [5].

In the *Cycle* scheduling, the thread TT processes the *element%number of threads*. If the number of elements is less or equal to the number of threads, this distribution is the same as of *Block* distribution.

ParalllelDynamic scheduler creates one task per element and a thread pool distributes the tasks by the threads.

Privatisation

Parallel maps allow parallel execution, but they may originate data races due to concurrent data accesses. The developer can use the Java mechanisms to avoid data races, however, in HPC problems, these mechanisms can introduce an unacceptable overhead. An alternative can be the thread private data.

To use this mechanism with the approach, the developer needs to specify the private fields in the domain model (figure 3.6). The developer adds *private* attributes into the *Annotated element*. The UML tool generates a new split that returns a collection where the private field is visible to a single thread[6], and shared fields can be accessed by all threads. The developer uses the original method, but the data access methods hide if the field is private or shared.

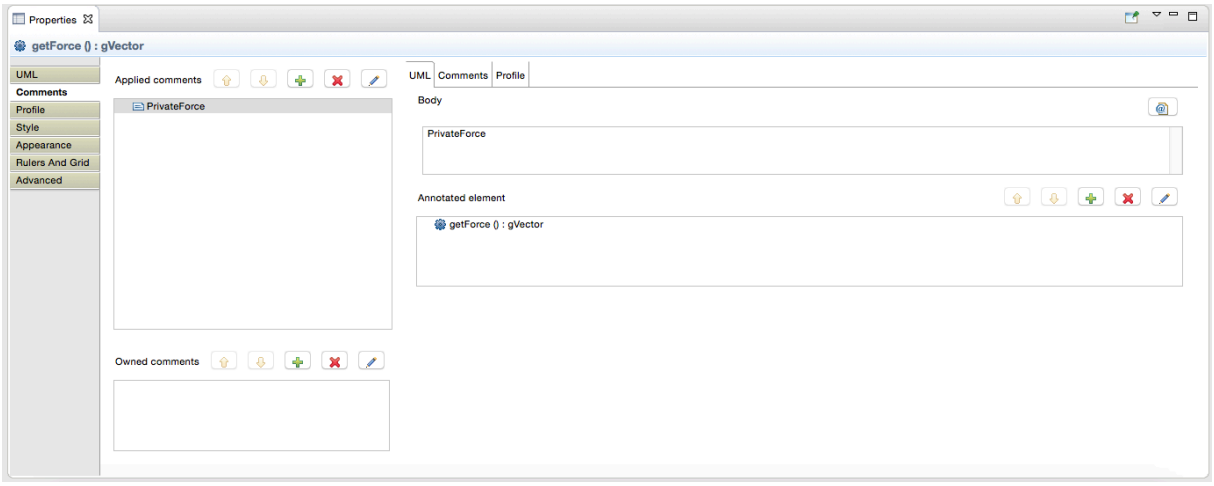


Figure 3.6: Example for privatisation data in MD case study

[5]The block X is processed by thread TT-X (TT is the total of threads, and the number of blocks must be equal to TT x 2).
[6]Private fields are initialised by default to 0.

The developer uses the new *split* which name is equal to the comment defined in the UML tool. At the end of processing, it is necessary to update the original collection. The developer must define the method for this purpose. This method receives two parameters: the object pointing to the private collection, and the object to access the original collection.

In the MD example, the developer needs a private *force* vector. The figure 3.6 shows how to create the new split that enables private *Force* vector in the UML tool. The developer uses this split, in the *gSplitMapJoin* (see listing 3.9), to enable private *Force* vector on each thread. Additionally, the developer creates and uses the *md::reduceMethod* to reduce the private fields.

```
@gSplitMapJoin(name = "PForce", map = "ParallelBlock", split = {"Virtual", "PrivateForce"},
    reduce = {"default","md::reduceMethod"})
```

Listing 3.9: Annotation example using a private collection in second argument.

Listing 3.10 show *md::reduceMethod*. The developer defines how the *c* must be added to the *ret* (after processing the ret value is copied to the original collection). In this case, the collection's values are reduced using the sum operation.

```
public static void reduceMethod(Particle c, Particle ret){
  ret.getForce().setX(c.getForce().getX() + ret.getForce().getX());
  ret.getForce().setY(c.getForce().getY() + ret.getForce().getY());
  ret.getForce().setZ(c.getForce().getZ() + ret.getForce().getZ());
}
```

Listing 3.10: Method reduce - sum all private values

Data sorting

Data sorting can be applied to the AoP collections. The consecutive collection elements are placed in successive memory positions[7], to improve spatial locality. The developer just calls the collection's *sort* (listing 3.11).

```
  collection.sort();
```

Listing 3.11: Using the sorting optimisation

[7] Exchange the order of objects in memory.

Other optimisations

Several optimisations can be implemented manually. In the context of this document, the optimisations already described have been implemented and tested. However, the developed methodology also simplifies other optimisations, improving the code legibility (it is possible to hide other optimisations within the *gCollection* framework).

An example is data alignment optimisation. Aligning collections in Java does not imply that the first collection element is aligned. The approach can align this element by leaving empty spaces and redefining the access methods (for element n, the methods can access the element n+alignment). Thus, the approach can transparently align the first element of the collection, which is essential for efficient vectorisation.

Another option is to use the same iterator to process multiple collections (similar to the *Zip* iterator implemented in *boost* [8]). Also, the approach allows adding a mechanism to join or split collections. The mechanism implementation is similar to the virtual collections.

3.3 Implementation

The approach relies on a set of tools to support certain development steps. Figure 3.7 shows the different tools that help developers writing programs using the proposed approach. The developer creates the domain application in the steps represented by blue boxes. The orange boxes are the optimisation specification steps performed by the developer. Finally, green boxes are the steps where the developed tools help the developer (one example is the collection generation). In the first step, the developer defines the domain entities that will be used to generate the required code library. In the next step, the UML tool generates the library to support the entities and collections. Then, the developer writes the domain code using the entities and collections of the generated code library. These three steps deliver a complete program that solves the domain problem, but the developed program is not yet optimised. In the following steps, the developer identifies optimisations to be applied in order to optimise the program execution time. For this, the developer uses the profiling tool (PAPIJ) or traditional benchmark/profiling tools. Subsequently, the developer selects the optimisations that will be applied. After that, the developer can go back to the previous step to analyse the impact of those optimisations.

[8] `https://www.boost.org/doc/libs/1_64_0/libs/iterator/doc/zip_iterator.html`

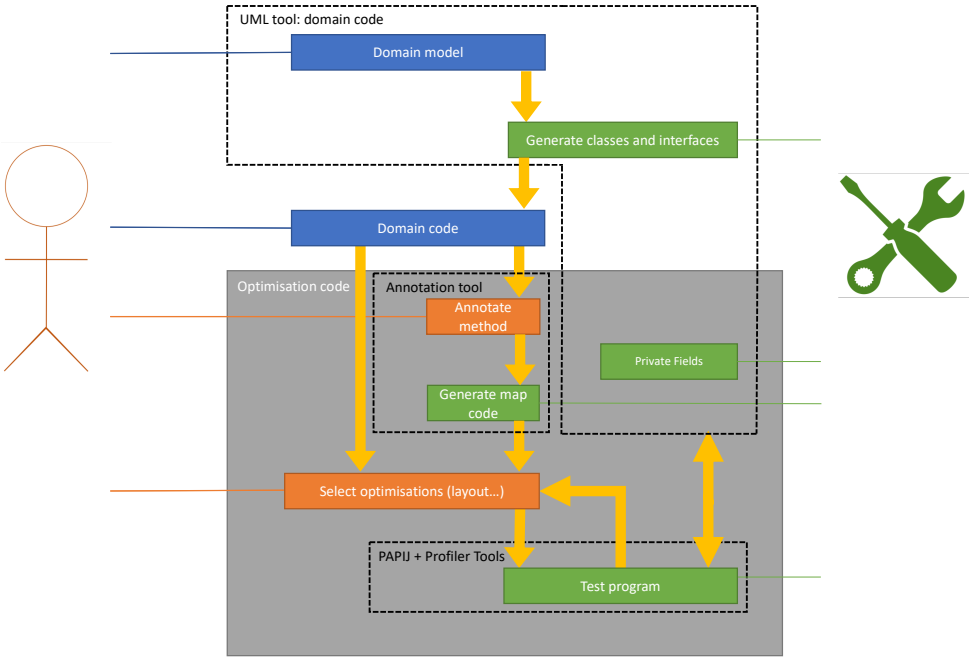


Figure 3.7: Development tools supporting the steps of the approach

3.3.1 Supporting tools

The approach has a set of tools that help the developer to build and optimise programs. The developed implementation includes three tools: UML tool, Annotation tool, and Profiler tool. The UML tool generates a set of interfaces and classes that implement collections for the problem domain. The Annotation tool creates classes to implement the domain decomposition. The Profiler tool enables the profiling of Java programs through annotations.

UML tool

The UML tool creates the interfaces and classes to support a collection of domain entities. These interfaces and classes are compatible with the Java *List* interface. The UML tool currently implements two layouts: AoP and SoA. The UML tool uses two eclipse plugins: the first allows the developer to specify the domain model and the second generates the classes and interfaces. The first plugin is *Papyrus*[9], a visual tool for developers, used to define the domain model (e.g. see figure 3.3). The approach is not dependent on this plugin, since it is possible to use other tools to specify the domain model. The second plugin is the

[9] `https://eclipse.org/papyrus/`

Acceleo that provides the *Acceleo Query Language* (AQL)[10] to specify model transformations in Java code.

We developed modules in *Acceleo* to generate the interface and classes that support collections and the domain API. The tool generates two different components: the collections API and the domain API. The collections API is domain-independent, so it can be a pre-defined package that the developer imports. However, we decided always to generate this package. The tool generates interfaces and classes to support the domain API based on the domain model.

The tool generates the code in five steps (figure 3.8)[11].

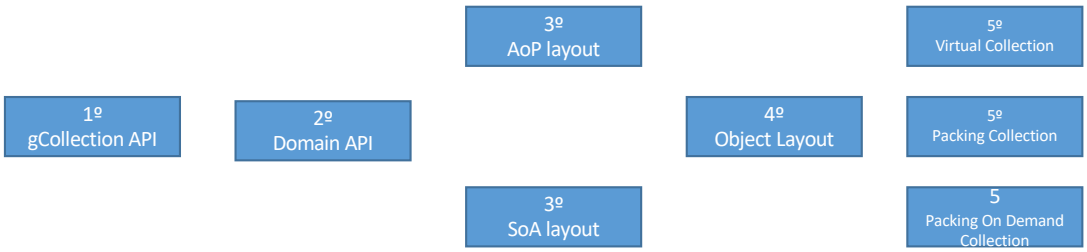


Figure 3.8: Steps to generate interfaces and classes

First, the tool generates interfaces to implement the collections API. The tool uses code templates to create those files. The step generates seven files that are placed in the *gCollection* package. The first is the interface *gCollection* that defines the API to access the collection. This interface extends the Java *List* interface. The *gCollection* interface also provides several higher-order functions implementations through the *default* methods or static methods (e.g., maps for *gSplitMapJoin*). The parallel map used in *gSplitMapJoin* has additional complexity to deal with concurrent access. Parallel *PackingOnDemand* creates a packing collection per thread, so each thread does not use data from another thread. The parallel *reduce* uses a similar approach to enable concurrent accesses. This step also generates the *join* method. *gSplitMapJoin* uses this method for the data privatisation technique. The method allows the developer to reduce the data defining only one simple method.

The package also contains *ParametersBiFunction*, *ParametersFunction*, *voidBiFunction* and *voidFunction* that extends the corresponding Java interface *Function* to accommodate several parameters configurations. The higher-order functions and the *gSplitMapJoin* mechanism require these interfaces.

This step also generates the *gIterator*. The class implements the iterator API and extends the Java iterators API: *Iterator*, *ListIterator*. All objects implement the interface *gCopy*. This interface has a simple method that enables to copy of the object (similar to the Java *clone*). Finally, *gException* is a class to throw exceptions in *gCollections*.

[10] `https://www.eclipse.org/acceleo/documentation/aql.html`

[11] The order of the steps is defined for this presentation, but it is possible to generate the classes in any order.

Secondly, the tool generates concrete domain interfaces and classes that do not depend on the layout. For each entity, the tool generates an interface, an abstract class, a concrete class (e.g., *realParticle*) and a *Factory* class. These interfaces and classes are available in the package defined by the developer in the domain model (figure 3.3). The interface has the same methods specified by the developer in the domain model. The developer can use the abstract class to implement additional methods of the entity. The tool converts the *get* and *set* into the class fields and generates those methods that implement the entity interface. The developer can use this concrete class in specific cases: initialise the *gCollection*; the layout AoP uses an array of this class; etc. Finally, the *Factory* class enables the developer to create a collection of the entities.

The third step generates two packages providing the concrete layout implementations: the *aop* package has classes to support the AoP layout, and the *soa* package supports the SoA layout. For each entity, there are two classes in each package: one to support the *gCollection*; another to support the iterator over the *gCollection*. The AoP collection is an array of the concrete class (generated in the first step). For the SoA collection, all fields of the entities are arrays of primitive types. The tool analyses the *get* methods and flattens the structures. It means, if the type is primitive, the tool adds an array of this type, otherwise the tool analyses the type recursively until it is primitive. For instance, in the MD case, the Particle interface has three *get*s that return *gVector*. The tool analyses the first *get* (*getPosition*), since it returns a *gVector* (non-primitive type), the tool analyses this object (*gVector*). *gVector* has three *get*s that return *doubles*. Thus, the tool adds three arrays to the collection Particle. Something similar happens to *getVelocity* and *getForce*. The listing 3.12 shows the fields definition of a *Particle* collection with AoP (grey color) and SoA (blue color) layouts.

```
public class gCollectionParticle implements gCollection<Particle>{

    public realParticle gCollection[];              public double PositionX [] ;
                                                    public double PositionY [] ;
                                                    public double PositionZ [] ;
                                                    public double VelocityX [] ;
                                                    public double VelocityY [] ;
                                                    public double VelocityZ [] ;
                                                    public double ForceX [] ;
                                                    public double ForceY [] ;
                                                    public double ForceZ [] ;

    int size = 0;
    int globalPosition=0;
(...)
```

Listing 3.12: Implementation of collection fields

The iterator implementation is specific for each layout. However, in both cases, the iterator is a pointer

to the concrete collection (e.g., *gCollectionParticle*) and has an integer to store the element position in the collection.

If an entity is composed of other entities an auxiliary object is generated, which provides a bridge to access that data. For instance for a *Particle*, the tool creates three classes: *gBridgePosition* provides access to the position, *gBridgeVelocity* to the velocity data and *gBridgeForce* to access to the force. The listing 3.13 shows how the developer accesses the value x in the position. The *getPosition* creates an object *gBridgePosition*. In the AoP, the *getX* accesses to the *Particle* and returns the *double*, using the *getPosition* and *getX*. For SoA, the *getX* returns a double from the *PositionX* array. In this specific case, it needs to create another object.

```
public gVector getPosition (){
   return new gBridgePosition(this);
}
```

a) Method to access the vector position

```
public double getX (){
   return this.gBridge.gCollection.gCollection        return this.gBridge.gCollection.PositionX
     [this.gBridge.position].getPosition().getX();        [this.gBridge.positionArray];
}
```

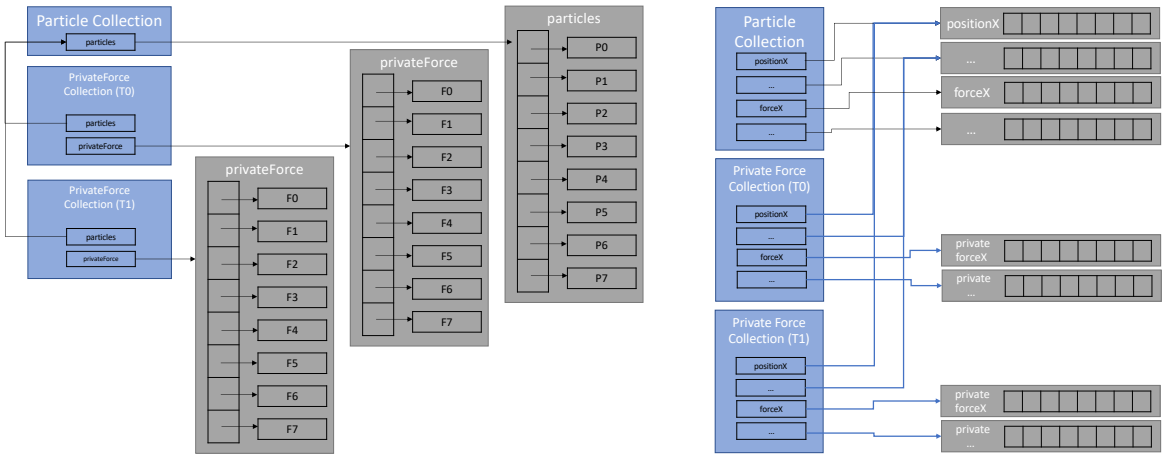
b) Method to access the coordinate X

Listing 3.13: Example of *gIterator* methods on composed entities

The tool also generates code to implement private data at this step. The tool creates a private collection (figure 3.9, the collection has eight elements and the example uses two threads). For the AoP layout (figure 3.9a), the private collection uses an additional array to save private fields. This array is composed by a new class with capacity to save all private data. For the SoA layout (figure 3.9b), the tools creates multiple arrays (one array for each private field). For all layouts, the tool extends the original *gCollection*, adding private collection class. The extended class returns a special iterator where shared fields use the original collection, and the private fields use the private arrays.

In the fourth step, the tool generates the classes to support a generic collection. In the approach, the *gCopy* is the most generic interface available (all objects implement the *gCopy* interface). This step generates two classes: one to implement the *gCopy* collection and the other is an iterator to the *gCopy* collection. The collection has an array of *gCopy* which implies the use of the AoP layout. The iterator is similar to other *gIterator* implementations (it has a reference to the *gCopy* collection and an integer to save the element position in the collection).

The tool uses this collection to implement the domain partition using only a generic implementation to support the partitions. So, it uses multiple partition levels. The number of elements in these collections



(a) AoP layout (b) SoA layout

Figure 3.9: Private fields implementation

is usually small, so the overhead due to the layout[12] is negligible in performance.

The last step generates the classes to implement domain partition. It generates three packages: one supports virtual collections, and the other two packages support the two variants of packing of collections (basic packing and lazy packing). The tool uses code templates to generate these packages. Each package has three classes: the first supports the collection of subcollections; the second is an iterator for that collection of subcollections; last is the subcollection implementation. The differences among those three packages are mostly in the implementation of the subcollection. *gCollectionVirtual*[13] (figure 3.10a) is a view from the original collection. The class has a reference to the original collection and two integers: one to the start position and the other to the end position of the subcollection. The methods that access the subcollection return a *gIterator* to the original collection, so this class adds low overhead.

The *gCollectionPacking*[14] class (figure 3.10b) has a collection of the same type as the original collection, but with a smaller size. This packing assumes the creation of all packing collections at the beginning, and at the end, updates the original collection. This option duplicates the collection footprint in memory (data replication). Moreover, it has an initial overhead since it creates all packing structures at the beginning.

PackingOnDemand reduces the initial and final cost to manage the subcollections and also reduces the size needed to save the data. All instances of the *gCollectionPackingOnDemand* use the same packing

[12]The use of a generic collection to implement all kinds of domain partitions implies the usage of the AoP layout.
[13]Class that contains one subcollection when the developer uses the *splitVirtual*.
[14]Class that contains one subcollection when the developer use *splitPacking*.

(a) Virtual partition (b) Packing partition

Figure 3.10: Domain decomposition implementation

collection to save the data. The data is packed into the subcollection before processing. When processing is complete, the data is rewritten in the original collection. For this purpose, the maps use the methods *read* and *write* available in interface *Parameters* [15]. If the map is parallel, it needs to create a packing collection for each thread, so each thread allocates a new collection to store the packing.

Annotation tool

The Annotation tool generates the code for domain decomposition, hiding implementation details from the developer. The tool uses the APT [16] tool available in Java to process the user-provided annotations.

The tool generates two methods (listing 3.14): the first has the same API as the original annotated method but places the method parameters into one *Parameters* class; the second receives that parameter class and calls the original method.

```
public static void force( gCollection<Particle> p1, gCollection<Particle> p2){
  PForce aux = new PForce(p1, p2);
  gCollection.mapParallelBlock(PForce::force, PForce::split, PForce::join, aux);
}

public static void force( Parameters parameters){
    //calls original method
    md.force( parameters.p1, parameters.p2);
}
```

Listing 3.14: Pseudo-code of generated methods (listing 3.9)

The tool also creates a split that decomposes the problem domain. For this purpose, it generates parameters collections *gCollection*. Split returns a collection of collections where each element represents

[15] *read* loads the data to packing and *write* store the data in the original collection.

[16] `https://docs.oracle.com/javase/7/docs/technotes/guides/apt/`

a subdomain. Finally, it aggregates the subdomains into *Parameter* collection.

Listing 3.15 shows a split method generated by the Annotation tool. The method starts by splitting the collections through the strategy specified by the developer. In this example, it uses a split virtual to divide the collection *p1*. The *p2* collection uses a specific split defined by the developer in the UML tool: the *splitThreadData* generates subcollections where some fields are private to each thread. In the UML tool, the developer defines the label *PrivateForce* which identifies this special split. The next step creates subdomains. In the loop body, it creates the subdomains. *PForce* class saves the data relative to each subdomain.

```
public static gCollection<PForce> split(Parameters obj){
  int numberblocks = Integer.getInteger("tile", 2);
  PForce parameters = (PForce) obj;

  //collections split
  gCollection<gCollection<Particle>> aux1 = parameters.p1.splitVirtual(numberblocks);
  gCollection<gCollection<Particle>> aux2 = parameters.p2.splitThreadData("PrivateForce",
      numberblocks);

  //create a new collections of parameters
  gCollection.gCollection<PForce> ret = new gCollection.Object.gCollectiongCopy(numberblocks);

  //creates all new parameters and put in the collection
  for(int i=0; i < numberblocks; i++){
        PForce newparameters =  new PForce( aux1.get(i), aux2.get(i));
    ret.set(newparameters, i);
  }
  return ret;
}
```

Listing 3.15: Split generated for example listing 3.9

The tool also generates the join (listing 3.16) that updates the original collection. This example uses the default methods to join the *p1* collection: the *p1* uses a *splitVirtual* so the join only returns the original collection since in this split the writes are performed directly in the original collection. For *p2*, it needs to join each private collection to the original collection. To do this, it calls the methods defined by the developer (listing 3.10) to join all private collections.

If the developer uses the packing option, the annotation tool generates similar code, replacing the split and reduce, with the equivalent versions. The *read* and *write* methods were created to support the *PackingOnDemand*. So before the method is called the data is loaded into the packing collection and after processing it is returned to the origin location. The tool generates both methods.

```
public static Parameters join(gCollection<PForce> obj){
  int numberblocks = obj.end().positionArray();
  PForce ret = new PForce(obj.get(0)._original_p1, obj.get(0)._original_p2);
  for(int i=0; i < numberblocks; i++){
    PForce collection = (PForce) obj.get(i);
    ((gCollection<Particle>) collection.p2).joinThreadData(md::joinPrivateForce, (gCollection<
        Particle>) ret.p2);
  }
  return ret;
}
```

Listing 3.16: Join generated for example listing 3.9

Profiler tool

The profiler tool measures the performance of annotated methods, allowing the developer to obtain some performance metrics from a specific method in a simple way. The tool has two operating modes. The first uses the Performance Application Programming Interface(PAPI)[17] [TJYD10] to obtain the hardware counters. The second uses the *currentTimeMillis()* to measure the execution time.

The profiling intercepts the specific methods to add the code for profiling. For this, we use AspectJ to intercept code in two points. The first point is the main method since it needs to start the PAPI library (for the first mode). This also adds the code to print the measured values at the end of the program execution. The second point intercepts the annotated method. At the beginning of the method, it adds the code to start to measurement. In the end, it stops the measurement and saves the values into a *HashMap*.

In both modes, the tool measures the values for each thread individually. Each measure adds the value in the *HashMap* whose key is the thread identifier. Thus, in the end, it prints the values obtained by each thread.

3.4 Tool limitations

The developed tools have several limitations that will be discussed in this section. The first limitation is the lack of support for polymorphism: entities in a collection should all belong to the same concrete class. Section 6.2 discusses how the limitation can be removed. The tool does not support this feature since the implementation is too complex, and polymorphism is not commonly used in scientific applications. Moreover, the solution could add overhead to the execution time.

The second limitation is that collections do not support null elements. This limitation arises from the support to the SoA layout, which does not support null elements by default. To support null elements,

[17] `https://icl.utk.edu/papi/`

this layout needs one boolean array that indicates if the values in the collection are valid or not [18]. This solution needs more space to save the collections and adds overhead (to test if one element is valid).

The third limitation is the current support of only two layouts: AoP and SoA. The tool does not support the AoS layout since Java does not directly support this layout. For this layout, there are two options: first, if all fields are of the same type it is possible to implement this layout with the manipulation of the indices; second, use of the Unsafe class that allows the writing of low-level code. Another option is the support for hybrid layouts, but these layouts are not attractive in our case studies.

A fourth limitation is a restriction on the type of methods where the mechanisms can be applied. The *gSplitMapJoin* does not support methods with a return value since, for the parallel execution, the return value requires an additional join of the values returned by all threads.

The last limitation discussed, the tool only supports simple data structures with one dimension (e.g., *List*). The Matrix Multiplication case study needs a new interface and classes. These interfaces and classes are generated manually using the same approach principles. However, JCF supports other types of structures: hash-map, stack, etc. In the case studies used in *Empirical Study of Usage and Performance of Java Collections* [CASL17] the List is used in 56% of the cases, the second most used is *HashMap* with 28%. Moreover, most scientific codes rely solely on simple arrays of data. For this reason, this work focused only on the *List* container.

[18] The problem is more complex with composed objects.

Chapter 4

Performance evaluation

This chapter evaluates the performance and programmability of the developed framework. The approach proposes a high-level API for HPC programs, so it is crucial to ensure minimal runtime overhead. One key point of the proposed approach is to provide mechanisms to test common data locality optimisations quickly. Thus, the performance tuning process is simplified, allowing fast development of an optimised version.

The performance is evaluated with five applications that test four different aspects. The first test uses two simple algorithms, which assess the basic programming interface and the base-line performance. The second evaluates the approach in the context of a Java framework for scientific applications whose code was already developed. The third application tests the *gSplitMapJoin* mechanism and the optimisation composition. The last test evaluates the extensibility of the framework with a new container (e.g. matrix).

The first test evaluates programming interface overhead in a simple context in order to also identify the causes of these overheads. For this, the evaluation uses two different algorithms whose results were also used for tuning the approach. This evaluation analyses several ways to iterate over collections: the traditional for, the *Java Iterator*, the *gIterators*, the higher-order function and *Java streams*. The first algorithm sums all the collection elements. However, the vectorisation is complex (Java 8 does not support the vectorisation of reducing operations). The second algorithm is daxpy. daxpy calculates the *y = alpha * x + y* operation for each element in the y collection, where alpha is a constant and x is the element in another collection. Thus, the daxpy successively operates over two double values from two distinct collections at the same time. This case study allows auto-vectorisation, so the goal is to analyse the iterators impact on vectorisation (i.e., the compatibility with auto-vectorisation). The use of Java streams is straightforward in the first algorithm. However, the second case is only implementable if the two collections are joined into a single collection. This evaluation can also assess the performance of parallel streams and higher-order

functions. The analysis of this case study was used to define the parallel map implementation/schedule to be used by default.

The second evaluation analyses, how the approach can be applied to a Java framework. For this, the evaluation uses the JECoLi framework. JECoLi is a Java framework for evolutionary computing that uses Object Oriented programming. The JECoLi uses a *List* of generic objects. In practice, if we analyse the usage examples provided in the framework repository the List can be of *Double*, *Integer* or *Boolean*. Our challenge is how to transform a generic list into several concrete lists. The concrete lists allow using the SoA layout to improve performance.

The third evaluation uses a more complex data structure. The base code comes from the JGF *moldyn* benchmark [BSW+00]. Moreover, this evaluation allows applying a large set of optimisations supported by the approach. The first optimisation is to improve the AoP layout by sorting objects. The second modifies the base AoP layout to a SoA layout. The third uses tiling optimisation so that each subdomain remains in the cache. Then, it uses parallel execution mechanisms, where it will test various scheduling options and the privatisation optimisation. In this case, it is also possible to make compositions of optimisations, as an example, using two tiling levels. Overall, this case tests several features of the *gSpliMapJoin* mechanism. To finish this evaluation, it will analyse the usage of a new API closer to that domain.

Finally, the last evaluation analyses the approach extensibility, with the calculation of matrix multiplication. In this case, the container is a matrix of doubles rather than a list, so, it requires new methods to access a matrix. This evaluation tests the packing mechanisms together with parallel execution to improve performance. This application uses a special matrix multiplication kernel developed in Java with low-level optimisations (e.g., a non-trivial low-level kernel). It also tests if the approach is compatible with these kinds of low-level optimisations.

4.1 Methodology

The section characterises the test environments and describes the methodologies used for each evaluation.

The results were obtained in the *SeARCH*[1] cluster using the *compute-662-6* node. The node has two *Intel Xeon E5-2695v2* processors with the *Ivy Bridge* architecture. Each processor has 12 cores and supports 24-threads. The processor supports Intel Turbo Boost technology which increases the processor frequency according to its design limits. As a rule, the frequency is increased to the maximum when only one core is heavily used. In this evaluation, the frequency is fixed at 2.4GHz, allowing the results to suffer less variation caused by several factors such as environmental temperature. Moreover, it allows

[1] `http://search6.di.uminho.pt/`

more accurate scalability analysis in parallel execution since the frequency does not decrease with more threads. The machine has three levels of cache memory. The first two levels are private to each core. The first level is divided into instructions and data cache and has an access cost of 4 or 5 cycles. Its size is 32KiB for instructions and 32KiB for data. The second level has a size of 256KiB and an access time of 12 cycles. The L3 cache is shared by all processor cores and has a total of 30MiB. Its access time is approximately 30 cycles. The node has two processors, so there is a total of 60MiB of L3 cache.

A personal machine is used to obtain the execution profile, using the VisualVM tool. The VisualVM can obtain the profile through two methods: Profiling and Sampling. The Profiling mode intercepts the method calls and measures execution time. The Sampling mode interrupts the execution every delta time to analyse the call stack. This evaluation uses the sampling mode to obtain the execution profile since it introduces less overhead in the analysis, and it does not restrain compiler optimisations.

The programs execution use the JVM from the OpenJDK 1.8.0 20 package. Additionally, the evaluation uses several JVM tuning parameters: -Xmx32G, -XX:LoopUnrollLimit=100, -XX:+UseCompressedOops and -XX:+UseNUMA. -Xmx defines the maximum size occupied by the heap. Some evaluations need more memory than the default value. -XX:LoopUnrollLimit makes unrolling optimisation more aggressive. -XX:+UseCompressedOops forces the use of compressoops which reduces the size occupied by pointers. -XX:ObjectAlignmentInBytes=32 forces objects to be aligned to 32 bytes. With -XX:+UseNUMA the compiler privileges the object's allocation in the memory closer to the processor.

The first evaluation studies the overheads of the approach at a low level. For this propose, the evaluation discards the first five executions of the algorithm since this discards the cost of loading data to fast memory and the JiT compile time of the program. The main method executes ten times to increase the problem granularity (for a better clock resolution). The program runs 100 times, and the values presented in this section are the median of values. This evaluation analyses the performance values computed per element since these allow comparison with theoretical values. The evaluation analyses the assembly instructions in order to understand the costs or benefits of each option.

The JECoLi evaluation studies the three algorithms that use the largest sized collections (collections with more than 1000 elements). Other algorithms provided in the JECoLi repository are not attractive for performance improvements since the collections are small.

MD evaluation compares the base JGF version with a base implementation using the proposed approach. After this comparison, the evaluation optimises the base implementation with the tools available. The evaluation analyses the impact of each optimisation on the code. Finally, the evaluation tests a new code version using a model of entities closer to the domain.

MM evaluation tests two distinct layouts: vector or array of arrays. The first analysis compares these

two versions to choose the best representation. After this step, the evaluation applies other optimisations available in the approach. Finally, the evaluation compares optimised versions with existing MM libraries for Java.

4.2 Programming interface

This section evaluates the base programming interface. For this, two simple algorithms are used to analyse and quantify the overheads of different programming interfaces. The first algorithm sums all elements in the collection. The second algorithm multiplies one constant with each element in one collection and adds this result to the corresponding element in another collection. These two simple algorithms allow obtaining performance counters per element that can be compared with the theoretical performance.

4.2.1 Sum

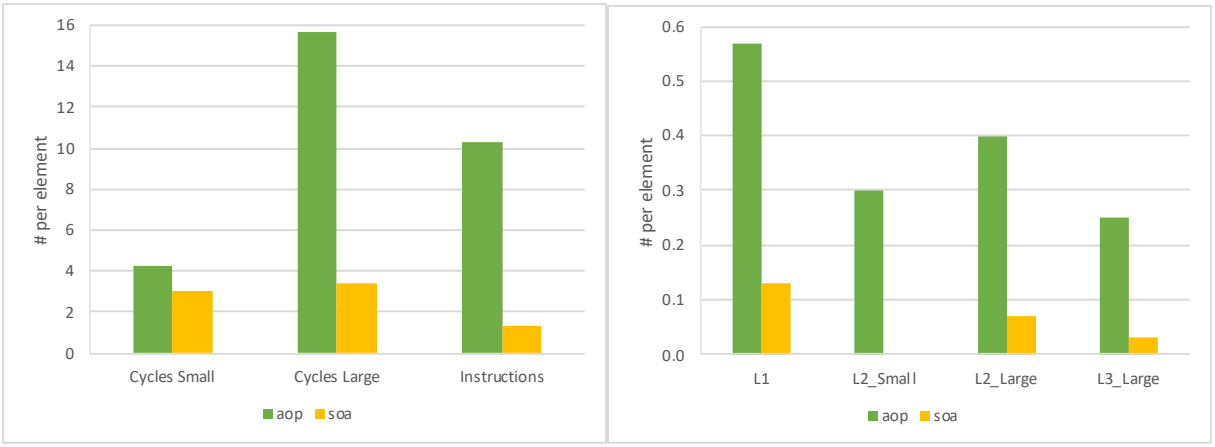
The *Sum* benchmark sums all values of a collection. To sum each value, it reads the value from the collection and adds this value to the result. In the most efficient version, the variable result remains in a register, and the value is only saved in the memory when the processing finishes. This case requires only one X86-64 instruction[2] to sum each element. This operation implies a memory read and one arithmetic operation.

The benchmark analyses the two layouts available in the approach: AoP and SoA. AoP uses an *ArrayList* from the JCF. The *ArrayList* is an array of objects, which are instances of the class *Double*. The developed framework implementation also uses an array of objects. However, the elements are instances of class *realDouble*. The SoA layout uses an array of primitive doubles.

The evaluation analyses two problems sizes: $6.4\text{x}10^5$ elements ($\sim$5MiB) and $5.12\text{x}10^7$ elements ($\sim$390MiB). In the AoP layout, each element is an instance of class *Double* or *realDouble* and consumes 32 bytes[3], so the effective size of the problem is $\sim$19MiB and $\sim$1562MiB. For all layouts, in the small size, the problem fits in the L3 cache, while in the large size, it requires accesses to main memory.

Figure 4.1a shows the impact of the layout. For the large size, the SoA layout is $\sim$4 times faster than the AoP layout (Cycles Large bars). The AoP layout requires more instructions since it performs two loads per element (load the object pointer and load the data object, see instructions bars in the figure). Moreover, the data access performance is worse for two reasons: the effective data size increases; and

[2] X86-64 instructions support one arithmetic operation and one load or store operation in one instruction.
[3] Objects are aligned to 32bytes.



(a) Time and instructions performance

(b) Memory access performance

Figure 4.1: Performance of AoP and SoA layout (Java version).

the elements are not contiguous in memory (less spatial locality).

The results presented in figure 4.1b show that L1 misses are close to the expected number of misses. The cache line has 64 bytes. In the SoA layout, each element takes 8 bytes. Thus, on a miss eight elements are loaded that results in 0.125 misses per element which is in line with the measured 0.13 per element. In the AoP layout, one object takes 32 bytes causing 0.5 misses to load the object. Additionally, it needs to read the pointer to the object from the array, and each pointer uses 4 bytes[4]. Each miss forces the load of 16 pointers (64bytes/4bytes) to the cache, resulting in 0.0625 misses per element. Adding the two values, the misses per element are 0.5625, which is close to the measured value of 0.57.

The small size has better performance (fewer cycles per element), since all elements, in the collection, are in the cache, so there are no accesses to the main memory (i.e., there are no L3 cache misses).

Table 4.1 summarises the tested implementations and presents their acronyms. The lines represent the modes used for processing collection, and the columns show the layouts and approaches (Java and *GasPar*). Java versions use distinct codes for each layout: AoP (see listing 4.1, green code) and SoA (see listing 4.1, grey code). *GasPar* uses a similar code to Java AoP[5] for both layouts: AoP and SoA (see listing 4.1, blue code).

The first code variant does not use iterators (listing 4.1a), it accesses the element by its index (*get(i)*). The evaluation tests this code with all layouts, which creates four versions (table 4.1 line *basic version*): *aop* uses an *ArrayList<Double>*; *soa* uses a primitive array of *double*; *gaop* uses the *GasPar* with the AoP

[4] *compressedoops* force the pointers to use only 32 bits.

[5] Not the same due to Java's autoboxing and unboxing.

	Java		GasPar	
	layout AoP	layout SoA	layout AoP	layout SoA
basic version	*aop*	*soa*	*gaop*	*gsoa*
Java iterators	*faop*	*fsoa*	*fgaop*	*fgsoa*
GasPar iterators	-	-	*ggaop*	*ggsoa*

Table 4.1: Acronym of each sum implementation

```
for( int i=0; i < collection.size(); i++) {
  result += collection.get(i);     result += collection[i];       result += collection.get(i).getValue();
}
```

a) basic version: *aop*, *soa*, *gaop* and *gsoa*

```
for(Double value: collection) { for(double value: collection){ for( gDouble value: collection) {
    result += value;                                            result += value.getValue();
}
or
Iterator it = collection.iterator();
while(it.hasNext()){
  result+=it.next();              //not support                result+=it.next().getValue();
}
```

*b)*Java iterators: *faop*, *fsoa*, *fgaop* and *fgsoa*

```
gIterator it = collection.begin();
for(;it.isless(collection.end()); it.inc()){
    result += ((gDouble) it).getValue();
}
```

c) GasPar iterators: *ggaop* and *ggsoa*

Listing 4.1: Different codes for sum

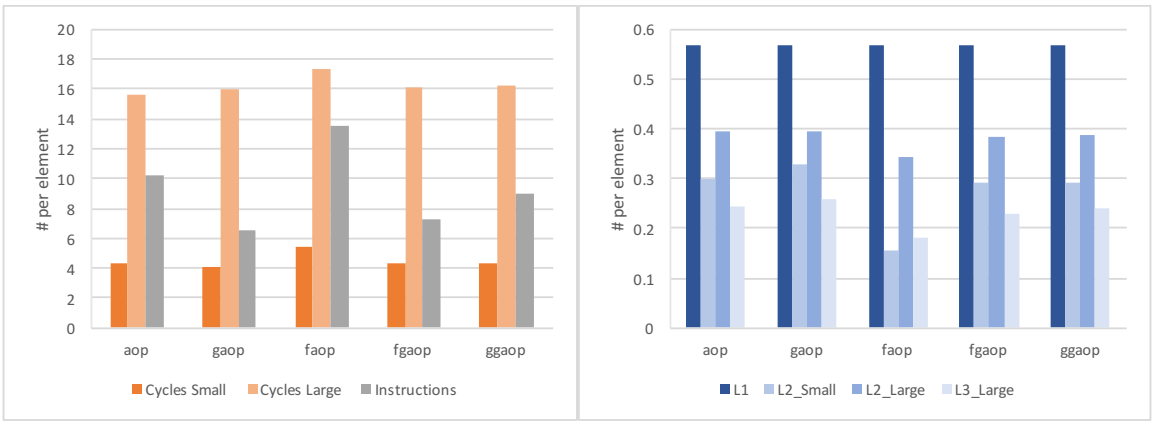
layout; *gsoa* uses the *GasPar* with the SoA layout.

The second variant uses Java iterators to access the elements (table 4.1 line Java iterators) and includes tests for the same layouts. The two codes in listing 4.1b produce the same bytecode, thus, the variants have the same performance[6]. In the *soa*, it is mandatory to use the first specification of listing 4.1b, since explicit iterators are not supported on arrays of primitive types. The second specification is more generic, as it allows the developer to navigate the collection by skipping elements, for instance.

The third variant uses the *GasPar* iterators (listing 4.1c). It supports the two layouts: *ggaop* for the AoP layout and *ggsoa* for the SoA layout (table 4.1 line *GasPar iterator*).

Figure 4.2 summarises the performance of the different AoP implementations. In general, the *GasPar*

[6] https://docs.oracle.com/javase/specs/jls/se8/html/jls-14.html#jls-14.14.1.2



(a) Time and instructions performance

(b) Memory access performance

Figure 4.2: Performance analysis for AoP layout

collections use fewer instructions to calculate the sum. Nevertheless, it does not imply a reduction in the execution time in the same proportion since the number of misses in L2 and L3 are higher (L1 misses are equal in all versions).

The use of iterators introduces a performance penalty (see *aop* vs *faop* and *gaop* vs *fgaop*) since it needs to store the index in memory (table 4.2). For the Java implementation, these costs are higher. One the other hand, *GasPar* collections have a smaller penalty: ∼0.75 additional instructions per element that imply ∼0.22 cycles per element.

The SoA layout generates simpler code in all versions (detailed table 7.1 in appendix). However, the *GasPar* has one more instruction on each loop body. It means that there is one more instruction per 16 elements resulting in equivalent performance.

	load object	test null	checkcast	store index	store result	unroll
aop	$$	$$	$$	-	$$	8
gaop	$$	-	-	-	$$	8
faop	$$	-	$$	$$$	$$	4
fgaop	$$	-	-	$$	$$	8
ggaop	$$	-	-	$$	$$	8
soa	-	-	-	-	$	16
gsoa	-	-	-	$	$	16

Table 4.2: Groups of instructions generated (assembler instructions in appendix listings 7.1, 7.2, 7.3 and 7.4)

Table 4.2 shows the purpose of the instructions generated that explain the differences in #I among the AoP implementations tested. The first column refers to if the version needs to load the object address

(element). In the examples (blue instructions in listing 4.2), typically, this operation requires three instructions. The first instruction accesses the array to load the object address. The second instruction copies the address into another register (this instruction has zero latency), and the last calculates the real object address[7]. These instructions are mandatory in the AoP layout but do not necessary in the SoA layout.

The "test null" column indicates if the version generates code to test if the object is null (yellow instructions). This test consists of two instructions. The first instruction tests if the pointer to the object is zero (test null), if so, the second instruction jumps to the null pointer exception handler code.

The third column shows what versions test the object type. The examples (red instructions) use three instructions. The first instruction copies the object type into a register. The second compares the register contents against the expected type. If the type is incorrect, the third instruction jumps to the exception handler code.

The "store index" column shows the cases where the object index is written to memory. If there is one write per element, the table shows symbol $$. The symbol $$$ indicates an additional overhead (e.g., the use of two variables per index). In these cases, Java has additional overhead since the *ArrayList* iterator also stores the last element index accessed. In the example (green instructions), there are four instructions for this operation. The first two operations calculate the new index. The third operation stores the index. The last operation stores the index in another memory location. *gsoa* writes the index in memory one time per loop body (symbol $).

The "store result" column identifies when the sum result is written to memory. "$$" writes once per element. "$" writes once per loop body. Typically, it needs one instruction for this operation (orange instructions).

The "unroll" column shows how many elements are processed in one loop body. This technique was described in section 2.3.1.

The table 4.2 explains why the *GasPar* collections generally use fewer instructions to calculate the sum. The approach does not generate the instructions to test the object type and test if the pointer is null since the JVM can identify that the collection uses one object type, and the objects are not null.

Java vs GasPar

The evaluation now focuses on a comparison of *GasPar* with the traditional approaches. Figure 4.3a shows that without iterators, *GasPar* has similar performance to Java.

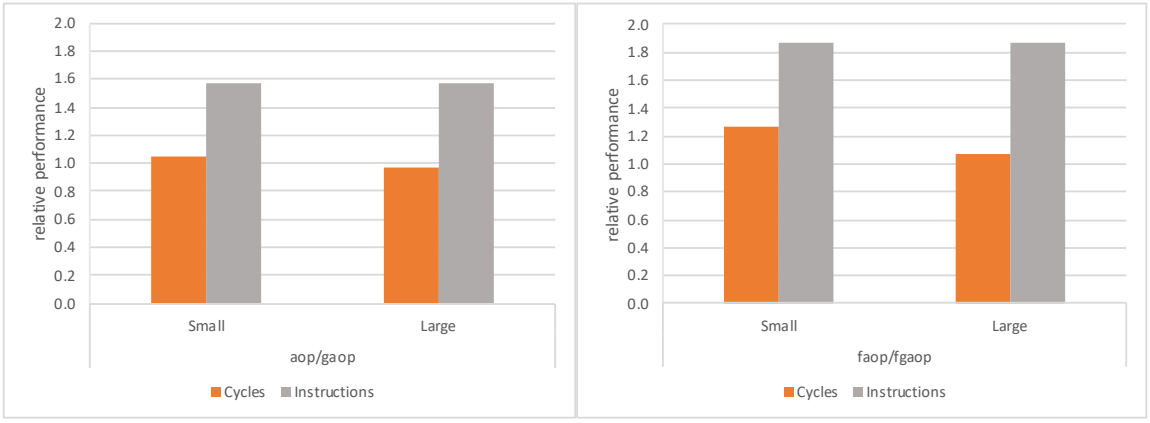
This evaluation has a bottleneck in the data access (the best version needs one memory access per

[7]The program uses the option *compressedoops* and aligns objects to 32 bytes. Thus, the last five digits of the memory address are zero. This option removes these bits, storing only 32 bits, but it needs to add the zeros to use the stored value.

```
//Assembly for aop
(...)
mov     0x18(%rdi,%rbp,4),%edx
(...)
mov     %rdx,%r8
shl     $0x5,%r8
test    %r8,%r8
je      0x00002b3f5c3fabd2
mov     0x8(%r8),%r9d
cmp     $0xed08,%r9d
jne     0x00002b3f5c3fac5b
(...)
vaddsd 0x10(%r8),%xmm0,%xmm0
vmovsd %xmm0,0x90(%rcx)
(...)
```

```
//Assembly for faop
(...)
mov     0x18(%rbx,%rbp,4),%esi
(...)
mov     %rsi,%r10
shl     $0x5,%r10
mov     %r8d,%ecx
add     $0x3,%ecx
mov     %ecx,0xc(%r9)
mov     0x8(%r10),%eax
cmp     $0xed08,%eax
jne     0x00002b7a60824652
mov     %ecx,0x10(%r9)
vaddsd 0x10(%r10),%xmm0,%xmm0
(...)
mov     $0x1d992fa280,%r10
vmovsd %xmm0,0x90(%r10)
(...)
```

Listing 4.2: Example of assembler instructions to calculate one element in *aop*



(a) Without iterators (b) Java iterators

Figure 4.3: Comparative performance of Java and *GasPar* (AoP layout)

each sum operation). For this reason, an improvement in the number of instructions has less impact on performance. In the small size, the *GasPar* has better performance since it removes the instructions to perform type checking and to test if the object exists (table 4.2, red and yellow columns). For the large size, the lower performance is due to a small increase in the number of misses in L3. The footprint in memory is not the same in both implementations since the allocation of objects is different: the *GasPar* creates all objects when creating the collection; the JCF version creates the objects later.

The *GasPar fgaop* has better performance than the pain Java *faop* due to a stronger reduction of number of instructions (figure 4.3b). This improvement is most noticed in the small size since there are no accesses to memory, so the reduction in the number of instructions reduction creates a bigger impact. In the large case, the number of instructions reduces in the same proportion, but there are accesses to the main memory (misses in L3). This memory bottleneck implies an equivalent performance.

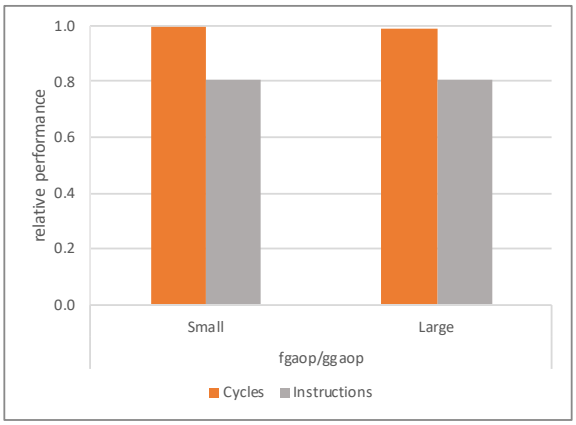


Figure 4.4: *GasPar* iterator performance

Figure 4.4 shows that the *GasPar* iterator has similar performance to Java-based iterator despite the increase in the number of instructions. These additional instructions have a minimal impact on performance since they are more two *mov* instructions (comparing listing 7.4 with listing 7.3 in appendix): one to copy a register to another register; the other moves a constant into a register).

We now focus on the SoA layout. The SoA layout uses only one instruction to load data and does not use instructions to test the object (check the object type and check if the object is not null). The several versions developed with SoA layout present similar performance, despite having a small difference in the number of instructions (see listing 4.3). The Java versions need 21 instructions to process 16 elements ($\sim$1.31 instructions per element), while the *GasPar* need 23 instructions ($\sim$1.44 instructions per element). The first additional instruction loads the result variable into a register, and the second additional instruction is generated due to an optimisation problem (writes to the same memory position in consecutive instructions).

Java streams vs GasPar

Java 8 introduced the stream API to simplify the processing of elements in collections. Java stream provides the method *sum* to sum all collection elements (listing 4.4). The AoP implementations need

```
//Assembly for SoA
(...)
vaddsd 0x10(%r11,%rbx,8),%xmm0,%xmm0
(...)
vmovsd %xmm0,0x90(%r10)
(...)
```

```
//Assembly for gSoA load and sum element
(...)
vaddsd 0x10(%r10,%r11,8),%xmm0,%xmm0
(...)
mov    $0x1f2abbdea0,%r8
vmovsd %xmm1,0x90(%r8)
vmovsd %xmm0,0x90(%r8)
(...)
```

Listing 4.3: Instructions need to calculate one element in versions *soa* and *fgSoA*

to convert the stream to *Doublestream*, with the *mapToDouble*. The operation converts the *Double* object sequence into a new sequence of primitive *double*. SoA layout only needs the *sum* method. These versions (from listing 4.4) are referred as *csaop* and *cssoa*. The *sum* uses the Kahan algorithm [Kah65] to reduce sum errors, but it increases the sum operation complexity from one arithmetic operation to four operations.

```
//Layout AoP (csaop)
result = collection.stream().mapToDouble( Double::doubleValue).sum();

//Layout SoA (cssoa)
result = Arrays.stream(collection).sum();
```

Listing 4.4: Java code for stream implementations

The evaluation used new versions to perform the sum operation (code is similiar to the listing 4.4)with *GasPar* collections: one with the AoP layout (*csgaop*) and another with the SoA layout (*csgsoa*). These two versions use the same code alike *csaop* [8]. Table 4.3 summarises the acronyms for these versions.

	Java		**GasPar**	
	layout AoP	layout SoA	layout AoP	layout SoA
stream based	csaop	cssoa	csgaop	csgsoa

Table 4.3: Acronym of each sum with compensation implementation

The figure 4.5 shows that the *GasPar* collections present a performance similar to Java in both layouts. The *csaop* improves slightly the performance since there is a decrease in the number of instructions. As the table 4.4 shows, the compiler removes the object checks (it does not test if it is null and its type). However, the *GasPar* needs to store the index in the memory, and the loop processes one element at a time. *csgsoa* has a performance similar to the *cssoa*.

The previous comparison did not allow us to make a conclusion about the efficiency of streams. For this purpose, we created new versions which implement the sum without the Kahan algorithm by using a

[8] Java *sum* method.

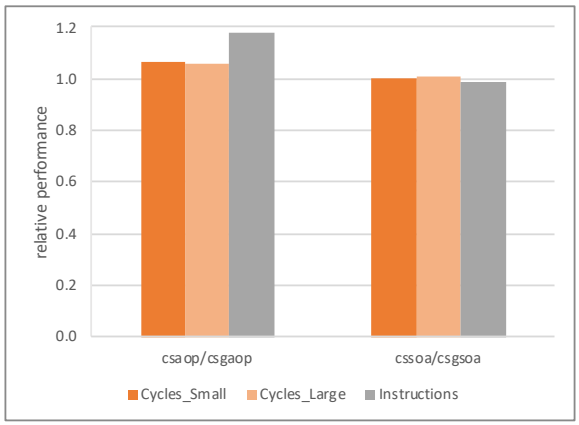


Figure 4.5: Relative performance the sum betweens Java and *GasPar*

	load object	test null	checkcast	store index	store result	unroll
csaop	$$	$$	$$	-	$$$	4
csgaop	$$	-	-	$	$$$	1

Table 4.4: Groups of instructions generated (assembler instructions in appendix listing 7.5)

reduce method. The figure 4.6 show the relative performance of the new versions for both layouts (*saop* uses streams with AoP layout and *ssoa* uses streams to process a *double* array) with the *GasPar* versions.

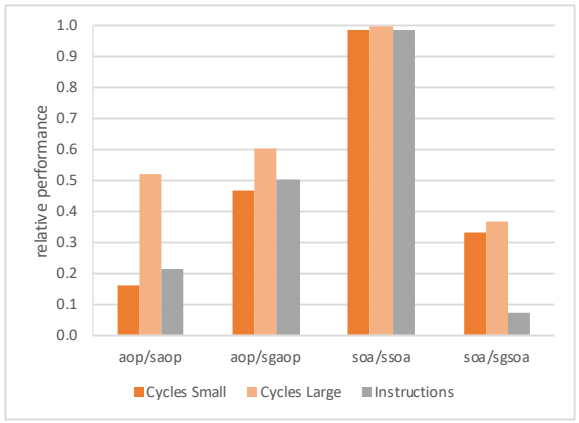


Figure 4.6: Relative performance with streams and *GasPar* versions with iterators

	load object	test null	checkcast	store index	store result	unroll
saop	$$	-	$$	$$	$$$	1
sgaop	$$	-	$$	$$	$$$	1
ssoa	-	-	-	-	$	16
sgsoa	$$	-	$$	$$	$$$	1

Table 4.5: Groups of instructions generated (assembler instructions in appendix listing 7.6)

The streams introduce significant overheads, except for the Java with SoA layout (figure 4.6). The decrease in performance is due to the number of instructions increases. Table 4.5 shows that the streams block the loop unrolling optimisation, also use more instructions for writing the result and the iterator position. For the SoA layout, the Java version has a similar performance since the code uses an optimised stream for primitive arrays (uses *Arrays.stream(soaCollection)*). *GasPar* use a generic stream for SoA layout that generates similar instructions to AoP versions.

Data sorting

The objects might be out of order in memory. For AoP, sorting can improve performance. Sorting changes the object positions in the collection for a better spatial locality.

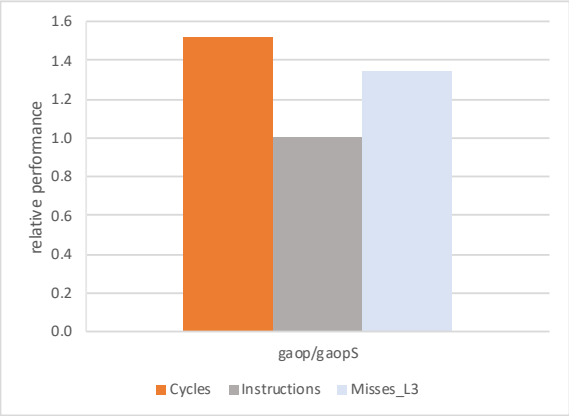


Figure 4.7: Gain of sorting collections

Figure 4.7 compares the *gaop* performance version with the sorting optimisation. Improving locality has no impact on the small size. Thus, the figure shows only results for the large size. In this evaluation, the sorting cost is not measured. For the large size, it increases performance 1.5 times due to the number of misses in the L3.

Parallel versions

The test in this case study evaluates the schedules available in *GasPar* in order to fine tune the library to deliver better performance, by default. Additionally, it compares the performance of the *GasPar* skeletons with Java streams. This test uses the *GasPar reduce* skeleton, which receives two parameters: the first is a method and the second is the return value.

GasPar versions use *reduceBlock* (listing 4.5) available in the *gCollection*. The default implementation uses the number of threads supported by the processor (48 threads). The threads can be managed in two different modes: explicitly creating new threads (*gXXXparallel*) or using a fork-join pool to process the tasks (*gXXXforkjoin*).

```
gDouble ret = new realgDouble(0);
collection.reduceParallelBlock(sum::gsum, ret);
result = ret.getValue();
```

Listing 4.5: Sum parallel version in *GasPar*

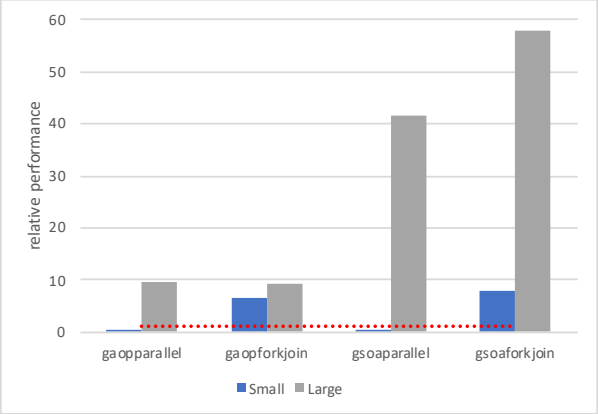


Figure 4.8: Analysis of gains of parallel sum in *GasPar*

Parallel execution and layout reduces ∼58 times the execution time (figure 4.8) when compared to base version (*aop*). The parallel versions with fork-join, in general, obtain the best performance. In the small size, there are speed-down in some versions due to the parallelism cost: the threads creation, and the data concurrency mechanism.

The next test compares *GasPar* with Java parallel streams (version sXXXparallel). In this case, the scheduling is the default. For Java, we use the *saop* version and change the sequential stream to a parallel stream.

Figure 4.9 show that the improvement is bigger with *GasPar* collections. The *saopparallel* version only reduces ∼4 times. The sequential version (*saop*) is five times slower than *aop*. When compared to the sequential stream version (*saop*) the speed-up is ∼7. *GasPar* skeletons have better performance.

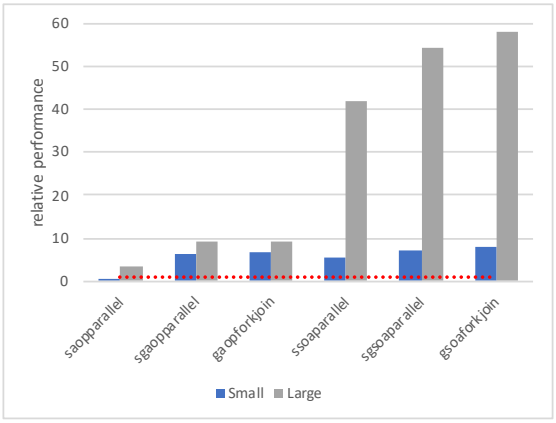


Figure 4.9: Compare *GasPar* with Java (parallel versions)

Summary

The evaluation analyses the overheads associated with the alternatives to access the elements of collections. The iterators added little overhead which makes them a valid alternative. The compiler generates fewer instructions for *GasPar* with the AoP layout. In these versions, the test of objects type and the null test is removed by the JVM. However, for the SoA layout, the *GasPar* collections generate more instructions but do not have a performance impact. The streams with objects introduced a substantial overhead which hinders its further use.

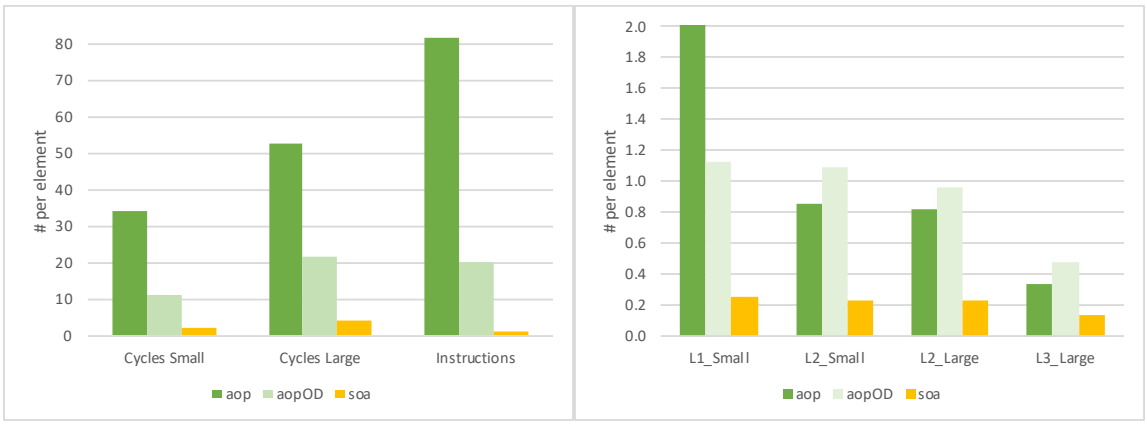
This evaluation uses several optimisations: data layout, sorting objects and parallel execution. In total, we reduce the runtime by ∼58 times for the large size and ∼8 times for the small size compared to the base version (*aop*).

4.2.2 daxpy

The second algorithm is the daxpy operation that multiplies the elements of array *x* with a value and adds the result to the elements of the array *y*.

This algorithm uses two collections and needs to access the same position in both collections. In Java, it is not possible to synchronise two iterators that limits the use of Java iterators. However, the iterators from JFC follow the natural order (from 0 to n). Therefore, they ensure access to the same index to both arrays when using two different iterators and going forward them at the same time. *GasPar* has the *sync* to synchronise the two iterators.

This algorithm uses the same sizes as the sum still the number of elements per collection was reduced to half. The evaluation uses the same layouts (AoP and SoA) with two different options: the first uses two



(a) Cycles and instructions performance

(b) Memory access performance

Figure 4.10: Performance of AoP and SoA layout[9]

collections to store the values; the second uses one collection with an object that holds two values, joining the two collections, which avoids using two iterators.

The daxpy algorithm requires two operations per element. The first multiplies the element of collection *x* with a scalar. The collection *x* elements are in memory, and the scalar value can be in a register. The second operation reads the element of collection *y* from memory, adds to the previous result and places the result into the collection *y*.

Once again, the results show that the SoA layout delivers better performance (figure 4.10). The *soa* has five times better performance than the *aopOD* since the number of instructions decreases, and the cache misses reduces in all cache levels. *aopOD* is a new version that improves the *aop* and is explained in the next paragraph. The number of instructions decreases for the same reasons as in the sum algorithm (e.g., remove instructions to load and test objects). Additionally, there are other reasons: AoP layout disallows vectorisation; the loop calculates fewer elements (there are more loop related instructions).

The *aop* variant uses Java Autoboxing[10] which causes a big impact on performance. The evaluation also uses an improved version, *aopOD*, which performs better (*aop* has half of the performance). The *aop* uses an ArrayList<Double>, whereas the *aopOD* uses an ArrayList<ODouble>. The *ODouble* class contains only a *double*, avoiding the Autoboxing and Unboxing. The version avoids the *Double* and forces a change to the object in the collection to update the value. On the other end, in the *aopOD*, to change the values in the collection, it needs to convert the *Double* into *double* and *double* into *Double* that creates a new *Double* object. Creating objects adds considerable overheads due to the memory allocation (activates

[9]The instructions and L1 misses are the same for small and large size and the L3 misses for small size are irrelevant. However in the *aop* for large size, the value of L1 cache misses is ∼1.71 misses per element.

[10]Autoboxing allows the developer to use Objects (*Integer*, *Double*,...) where the primitive types are expected and vice versa.

the garbage collector).

The L1 caches misses, in this case, present the expected values. The sum algorithm of the previous section needs one memory read to calculate one element: AoP layout has 0.5625 misses per element, and SoA layout has 0.125. The daxpy needs two memory reads to calculate one element, so this algorithm shows the double of misses per element: *aop* has 1.125 misses per element, and *soa* has 0.25. The measured value is ∼1.13 for AoP layout, and ∼0.25 for SoA layout (figure 4.10b).

Java vs GasPar

This section compares the performance between Java and *GasPar*. Table 4.6 shows the acronyms to identify each version. The *aop* is not included since it has poor performance.

	Java		**GasPar**	
	layout AoP	layout SoA	layout AoP	layout SoA
without iterator	*aopOD*	*soa*	*gaop*	*gsoa*
Java-based iterator	*faopOD*	-	*fgaop*	*fgsoa*
GasPar iterator	-	-	*ggaop*	*ggsoa*

Table 4.6: Acronym of each daxpy implementation

Table 4.7 shows that the *GasPar* reduces the number of instructions to test the object. On the other hand, the versions with iterators write the index values in memory, and the JiT does not apply the unroll optimisation in all versions with iterators. The table contains similar columns that represent the same kinds of instructions used in the sum evaluation. However, the column "store result" is not included since the algorithm needs to write the result always in the memory. Additionally, the table has a new column that shows the alternatives with vectorial instructions.

	load object	test null	checkcast	store position	vectorial instructions	unroll
aopOD	2x($$)	2x($$)	2x($$)	-	-	4
gaop	2x($$)	$$	-	-	-	4
faopOD	2x($$)	-	2x($$)	2x($$$)	-	1
fgaop	2x($$)	-	-	$$$	-	1
ggaop	2x($$$)	-	-	2x($$)	-	1
soa	-	-	-	-	+	4x4

Table 4.7: Groups of instructions generated

Figure 4.11 and figure 4.12 shows the performance for all versions present in table 4.7. The number of instructions is lower in the *GasPar* versions, but it does not improve the performance. The data access has more impact on performance for this algorithm since it needs two values per element. As in the sum, the performance difference between *GasPar* and Java is explained by the collection allocation. Once again,

the Java collection allocation is better for the large size. However, in the small size, the *GasPar* has better performance due to has fewer instructions.

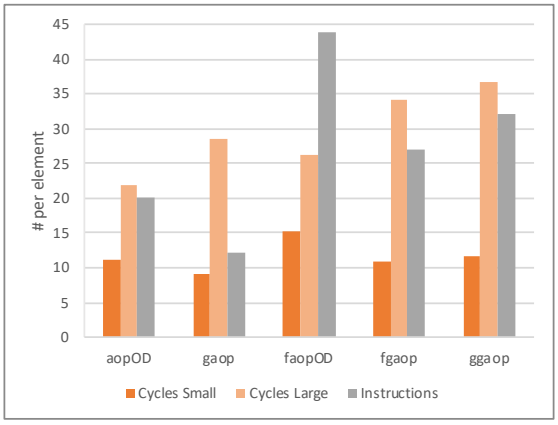


Figure 4.11: Performance analysis for codes in table 4.6 (AoP layout)

The iterators have a bigger impact on performance in this case study. The algorithm needs two iterators, and the compiler does not apply some optimisations (number of instructions increases). In the sum case study, the iterator is kept in a register, but, in this algorithm, the iterator is a variable stored in memory (e.g., reads and writes to the variable originate load and store instructions).

In this algorithm, with the SoA layout (figure 4.12), the *GasPar* and Java have the same performance. The *GasPar* also makes it possible to use iterators with this layout (it is not possible in Java). However, the iterator has a low performance since the number of instructions increases.

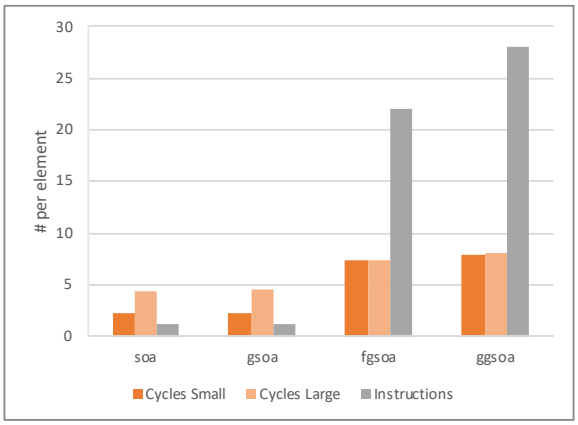


Figure 4.12: Performance analysis for codes presented in table 4.6 (SoA layout)

Note that in figure 4.12, for the SoA layout, the number of instructions per element is 1.25, which is lower than the number of the operation per element (the daxpy requires two instructions). It is possi-

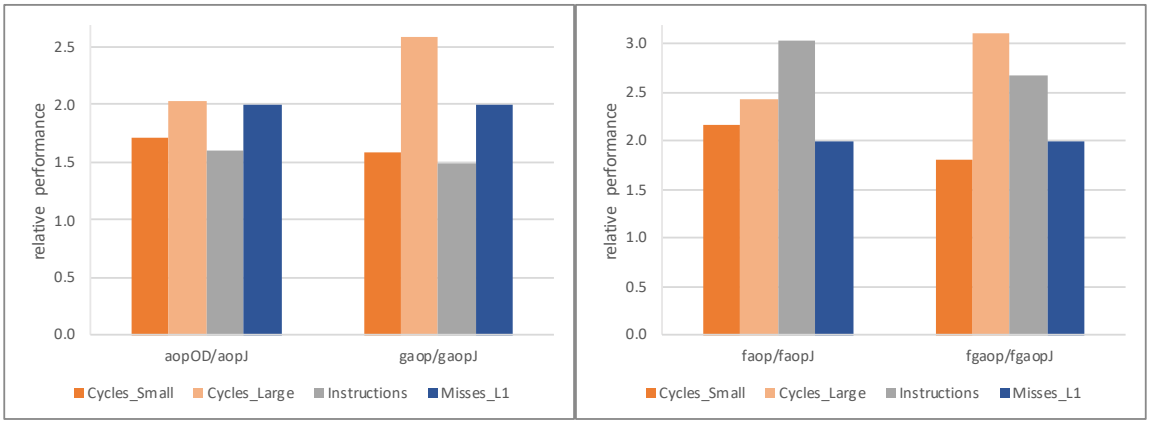
ble since the compiler uses vector instructions, so the processor deals with four elements in the same instruction.

The next test joins the collections where one element contains the two values. Table 4.8 presents the acronyms for these versions. Java does not support this alternative with the SoA layout.

The new layout has two advantages: first, it uses a unique iterator, and second, it reduces the footprint in memory for AoP layout. When using this layout AoP with a single collection, the real problem size reduces to half. One Java objects support two double with minimal objects size (objects are alignment the 32 bits).

	Java		**GasPar**	
	layout AoP	layout SoA	layout AoP	layout SoA
without iterator	*aopJ*	-	*gaopJ*	*gsoaJ*
Java-based iterator	*faopJ*	-	*fgaopJ*	*fgsoaJ*
GasPar iterator	-	-	*ggaopJ*	*ggsoaJ*

Table 4.8: Acronym of each daxyp with joint collection



(a) Without iterators (b) Java iterators

Figure 4.13: Relative performance between two collections and join collections for AoP layout

The figure 4.13 compares the performance of this layout to *aopOD*. As expected, the performance is better due to reductions in the instructions and cache misses. The number of instructions reduces $\sim$1.5 times (figure 4.13a) for the versions without iterators. The operations to load the object pointer reduces to half (compare table 4.7 to table 4.9). In the versions with Java iterators, the impact is considerable (figure 4.13b). The compiler reduces the number of instructions needed to read values from x and y collections. The *faopJ* enables the unroll optimisation, which further decreases the number of instructions. The number of misses also reduces since the real size of the problem is reduced to half.

	load object	test null	checkcast	store position	vectorial instructions	unroll
aopJ	$$	$$	$$	-	-	4
gaopJ	$$	-	-	-	-	8
faopJ	$$	-	$$	$$$	-	4
fgaopJ	$$	-	-	$$	-	8

Table 4.9: Groups of instructions generated

Regarding the layout SoA, the results are not presented as their performance is similar to the *soa* (see table 7.2 in appendix). The compiler applies optimisations that produce the code similar to the *soa* version. There is only a small increase in the number of instructions due to a store of the iterator value in memory.

In all layouts, there is a significant improvement in performance since these versions only use one iterator.

Streams

We use the join collection optimisation to assess the performance with Java streams since Java streams do not allow iterating over two collections simultaneously. Moreover, in Java, the SoA layout is not compatible with streams since it would need two collections. Therefore, the version named *saopJ* is the *aopJ* that uses the stream mechanisms to iterate over the collection.

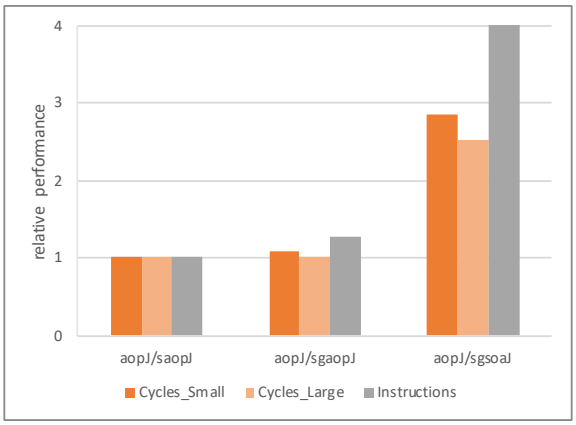


Figure 4.14: Relative performance between *GasPar* and streams

The use of streams does not add overhead in this case. In the AoP layout, the Java version and *GasPar* present the same performance. The *GasPar* version (*sgaopJ*) reduces the number of instructions, but the performance is similar. In *soa*, the improvement in the number of instructions is due to the layout (∼9 times, the graph is cropped to 4x).

Parallel versions

To conclude this analysis, the algorithm is executed in parallel. *GasPar* supports several implementations of the map pattern. In this evaluation, we use the *mapParallelBlock* to calculate the daxpy in parallel (listing 4.6). The evaluation uses the same number of threads as the number of threads support by the processor (48 threads). The threads can be created in two ways: the map method always creates new threads (*ggXXXparallel*) or uses the fork-join to process the tasks (*ggXXXforkjoin*). The Java version uses parallel streams.

```
gaopall.mapParallelBlock( a ->  a.setY(  alpha *  a.getX() + a.getY() ));
```

Listing 4.6: Parallel version in *GasPar* framework

Figure 4.15 compares two different parallel alternatives. Once again, the best results use fork-join to execute the tasks. For this reason, it will be used by default in *GasPar* framework.

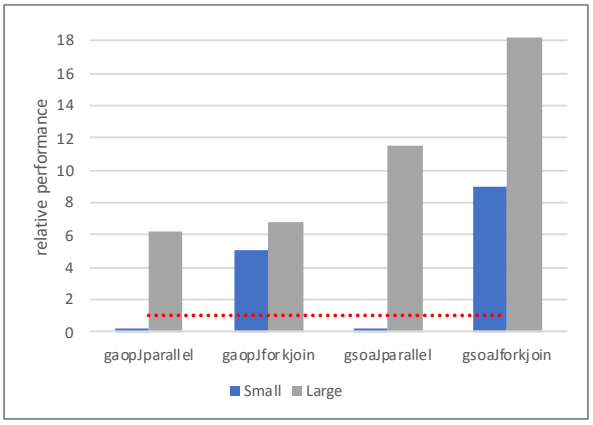


Figure 4.15: Analysis to parallel gains in *GasPar*

In this algorithm, the parallelism and data layout provide an improvement of $\sim$18 times, in the best case (figure 4.16). It needs more data to calculate an element, and this limits the speed-up. In this case, Java only allows implementation with streams with the AoP layout. Our approach allows more alternatives to implement parallel versions: use SoA layout and use the two collections. The two collections alternative shows a lower performance due to the need to use two iterators. The Java versions perform worse than using our collections.

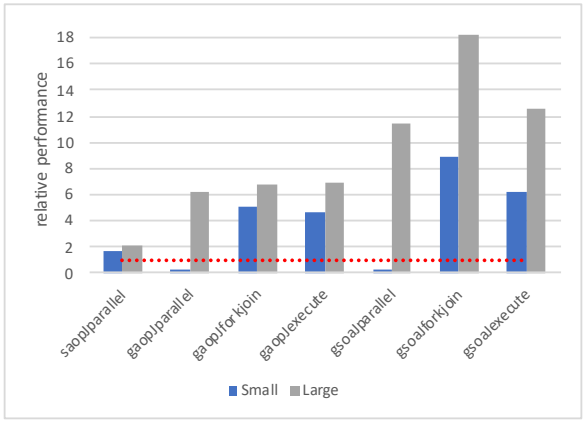


Figure 4.16: Relative performance between *GasPar* and streams

Summary

The evaluation of this algorithm tested again multiples alternatives to access the elements. In this case, the collection values are rewritten that cause Autoboxing and Unboxing of the values. Autoboxing/Unboxing creates additional complexity in the execution that introduces a significant performance cost. Additionally, we compare the use of two collections with a single collection. The two collections have an additional cost caused by the object load and the usage of two iterators.

For the SoA layout, Java only allows the use of the two collections with index accesses (*get(i)*, it does not allow using iterators). Our approach allowed us to create all alternatives for both layouts.

In this case, the optimisations provide an improvement of 18 times (layout change improves in $\sim$5 times, and the parallel execution improves in $\sim$3.5 times). Java versions only allowed a performance gain of $\sim$2.

4.3 Java framework - Java Evolutionary Computation Library

The Java Evolutionary Computation Library (JECoLi) [EMR09] is a framework to implement meta-heuristic optimisation algorithms. The main focus of the framework is Genetic and Evolutionary Computation. Frameworks in this field provide a set of algorithms to improve a set of solutions for some generations (i.e., iterations). Within a generation, some solutions are mutated/combined to find better solutions to the problem at hand. Typically, the most demanding of the computer power is the solution evaluation (i.e., calculate its fitness).

According to the authors, JECoLi uses OOP concepts to provide: flexibility, usability, adaptability,

modularity, extensibility and transparency. Therefore, it relies on Java collections and their intrinsic AoP layout. After analysing the JECoLi, we decided to change the *LinearRepresentation* implementation, which is one key class provided by the framework to represent a solution. We changed the *ArrayList* used in the implementation to a *gCollection*. The JECoLi class uses the *ArrayList* of objects that are a generics-based approach to represent a solution composed of a sequence of values (e.g., a sequence of bits). Our collections support the concrete objects that are generated from the domain model improving the performance.

The JECoLi repository has several examples illustrating the framework usage. This evaluation changes the *LinearRepresentation* that is used in all examples. The examples use collections of several types: *Double*, *Integer*, and *Boolean*. First, the evaluation uses *GasPar* with generic objects, in this case a *gCopy* collection. The solution allows us to support all examples. However, this solution can not enable SoA layout. Thus, the solution has a low performance. Below we optimise this first solution to create specific collections. There are three *GasPar* collections: a collection that provides *double*, another provides *int*, and finally one provides *boolean*. When, a *LinearRepresentation* is created, it is selected the concrete the *GasPar* collection to use.

Using *gCollection* introduces a restriction that does not exist with *ArrayList*: objects must be accessed only by *get* and *set* methods. However, Java has classes wrapping primitive types. These objects are special and can be accessed like primitive type objects (using the Unboxing)[11]. Thus, there is a need to rewrite the code to access the data through those methods. The listing 4.7 shows an example of the transformations required to use the *GasPar* collections. This case needs to add the *getValue* to read the boolean of the *gBoolean* class.

Finally, JECoLi objects implement features from other interfaces, such as *Comparable*. Therefore *GasPar* objects implement those features by extending the required interfaces and implementing the code with default methods.

This evaluation allows us to test how the approach works with frameworks that have already been developed. All examples provided can use *GasPar* collections. However, some examples use small collections, which is not suited to analyse the performance. Thus, the criterion was to analyse examples with collections with more than 1000 elements. Looking at the table 4.10, there are three cases under these conditions. All these cases use a collection of *Booleans* which in our approach was adapted to a *gBoolean*. The remaining cases have small collections, which make the collection layout irrelevant for performance (there is no impact on performance).

[11] In this case, Autoboxing and Unboxing do not introduce the same penalty of the daxpy since this case does not use Autoboxing.

```
protected int countOnes(ILinearRepresentation<Boolean> genomeRepresentation){
  int countOneValues = 0;
  for(int i = 0;i < genomeRepresentation.getNumberOfElements();i++)
    if(genomeRepresentation.getElementAt(i))  countOneValues++;
  return countOneValues;
}
```

a) Original version

```
protected int countOnes(ILinearRepresentation<gBoolean> genomeRepresentation) {
  int countOneValues = 0;

  for(int i = 0;i < genomeRepresentation.getNumberOfElements();i++)
    if(genomeRepresentation.getElementAt(i).getValue())  countOneValues++;
  return countOneValues;
}
```

b) GasPar Version

Listing 4.7: Code of countones: Original vs *GasPar* Collections

Package	Class	Data Type	Data size
countones	CountOnesCAGATest	Boolean	1000
	CountOnesEATest	Boolean	10000
knapsacking	EAknapsacking	-	
motifs	EAMotifs	Integer	5
	ProcuraMotifs	-	
	SeqMotifs	-	
multiobjective.countones	CountOnesMOSATest	Boolean	100
	CountOnesNSGAIITest	Boolean	100
	CountOnesSPEA2Test	Boolean	5000
	CountOnesSPEAMEMETest	Boolean	100
multiobjective.fonseca	FonsecaSPEA2Test	Double	3
multiobjective.kursawe	KursaweESPUMOSATest	Double	3
	KursaweMOSATest	Double	3
	KursaweSPEA2ArchiveTest	Double	3
	KursaweSPEA2Test	Double	3
	KursaweSPEAMEMETest	Double	3
multiobjective.schaffer	SchafferSPEA2Test	Double	1
multiobjective.wrapper	SAMOGenericTest	Double	3
numericalopt	EANumericalOptimization	Double	3
targetlist	EATargetList	Integer	50
	EATspOrdinal	-	

Table 4.10: JECoLi examples

All case studies are for the *countOnes* algorithm optimisation. The *countOnes* optimisation finds the best solution for a problem where the optimal solution has all gnomes set to true. The *CountOnesCAGA* uses the *CellularGeneticAlgorithm* to find the best solution. The solution has 1000 gnomes, and the process is repeated for 1000 generations. The *CountOnesEA* evaluation starts with 10 random solutions and the process stops after 100 000 generations. This case uses *EvolutionaryAlgorithm* to find the best solution. *CountOnesSPEA2Test* uses *SPEA2* algorithm to find the solutions. In this case, the population has 250 individuals, and the genome contains 5000 elements. The search stops after 500 generations.

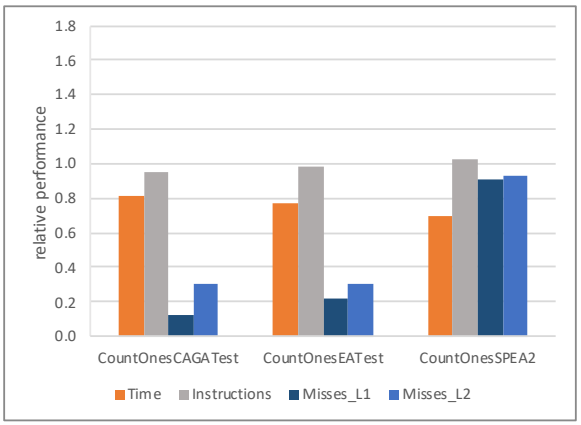


Figure 4.17: Relative performance between Original code and *GasPar* generic collection

The first evaluation compares the original JECoLi version with generic *GasPar* collection. Figure 4.17 shows a performance decrease in all cases. As mentioned, the initial solution allows us to execute the code in the framework context. Performance decreases since data accesses are less efficient.

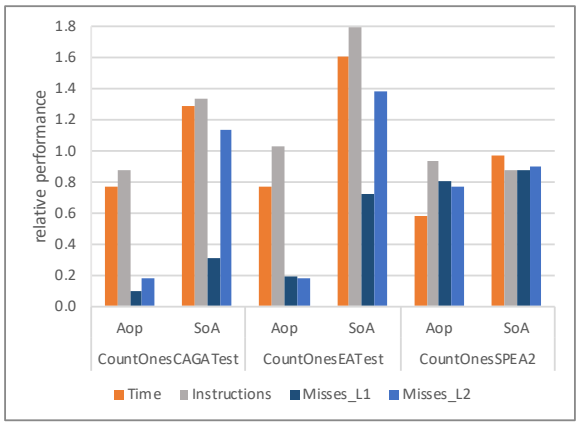


Figure 4.18: Relative performance between Original code and *GasPar* collection

Figure 4.18 shows that our approach has better performance when the developer uses the SoA layout since the number of instructions reduces, and the data access performance is better. The number of instructions reduces for two reasons: instructions of load reduce (SoA layout avoids one load per access) and more Java optimisations are applied. The AoP layout has a slight lower performance than the generic solution. In practice, the layout is identical, with small differences. First, collection allocation is different. The generic solution uses the element previously created and inserts it into the collection. In contrast, a concrete solution copies the data to the collection.

In the two previous examples, our collections have a composition relationship with *LinearRepresen-*

tation. That is, *LinearRepresentation* has a *gCollection*. Another solution uses inheritance by making *LinearRepresentation* a *gCollection*. The evaluation analyses this option for the SoA layout. This approach limits the *LinearRepresentation* to one single layout (the layout is defined at compile time and not during runtime when the collection is created). Thus, *LinearRepresentation* extends either a *gCollection* with AoP layout or a *gCollection* with SoA layout. The layout is a set in the *LinearRrepresentation* code, unlike the other option where it can be a program parameter.

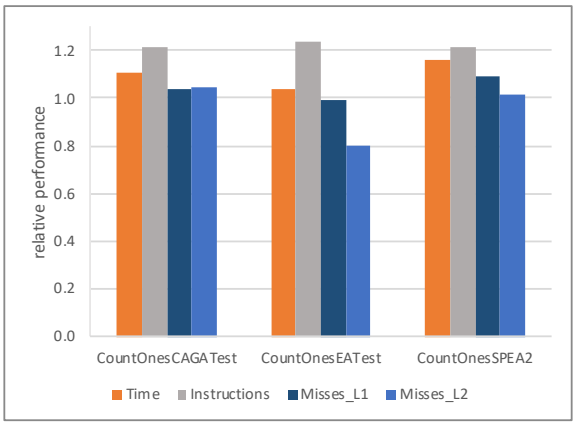


Figure 4.19: Relative performance between SoA versions and using inheritance

The figure 4.19 shows the improvement for the inheritance. In this case, the collection accesses are more efficient and decreases the number of instructions. The number of misses increases since the stress over memory is higher (the number of accesses with a time-frame are higher).

Summary

The proposed approach was able to improve JECoLi framework performance. The programming cost to include the *GasPar* approach in the JECoLi was one day. Thus, we can improve the legacy code performance (such as JECoLi) with a low development cost.

In this kind of frameworks, polymorphism is essential to allow its use in multiple cases. In this case, a mixed approach is created: using a generic collection to support polymorphism, and, in a second phase, concrete collections are created to improve performance in some case studies. As a result, it was possible to maintain the framework properties (flexibility, usability, adaptability, modularity, extensibility and transparency) allowing to improve performance.

4.4 Testing mechanisms - Molecular dynamics simulation

This evaluation analyses the approach in four new aspects: use of structured entities, the *gSplitMapJoin* mechanism, thread private data and composition optimisations. The case study deals with a collection of particles. A particle can be composed of other data structures. Thus, the evaluation explores two different alternative data structures to represent a particle. The molecular dynamics simulation (MD) can be optimised by decomposing the problem into subproblems. So, the evaluation tests the *gSplitMapJoin* mechanism to partition the problem in order to improve the performance. Parallel execution in this case study requires data access control in order to avoid data races. The evaluation uses the thread private data mechanism, available in *gSplitMapJoin*, to deal with the data race. Finally, the evaluation tests *gSpliMapJoin* with two levels (tiling with two levels, and tiling+parallelism) to evaluate the composition mechanisms.

The MD is an iterative process. Within an iteration, it moves the particles, advances the time, calculates the forces among particles and calculates the velocity of each particle. The process is repeated again for the number of iterations defined by the user.

The simulation uses a particles collection. Particles have attributes such as the position, velocity and force. Each attribute represents a 3D point in Cartesian space.

The evaluation uses the MolDyn benchmark available in the Java Grand Forum. The MolDyn simulates the interactions between Argon atoms. These atoms have Van der Waals interactions that use Leonard-Jones potential. The simulation space is a cube with periodic boundary conditions [BSW+00]. It means that when a particle moves outside the domain, it renters on the opposite side.

The code is organised in two main classes: *md* and *Particle*. The *md* class represents the simulation and contains a particles collection. The *md* class initialises the particles: initialises the positions, and generates the initial velocities. The force values have 0 value at the start. The *Particle* class has nine *double* that represent the position, velocities and force.

4.4.1 Applying the approach

To use *Gaspar* approach, the developer starts by creating the domain model in the tool. The domain model present in the JGF Moldyn has only the Particle concept. Particle is represented by a Java class and, in the approach, it is transformed into an interface. This interface has the *get*s and *set*s that access the nine *double* (figure 4.20). The interface enables the change of the layout.

After completing the domain model, the tool generates all code needed for the collections where both AoP and SoA layouts are available.

Figure 4.20: Domain model of a *Particle* (in the UML tool)

Then it is necessary to adapt the original code to use the *gCollection* interface. The listing 4.8 shows the changes required:

listing 4.8 a) the original code uses a particle array which is replaced by a *gCollection<Particle>*.

listing 4.8 b) create the collection, using the factory available in the *GasPar*. For this, create a factory and then create a new collection, by calling the *creategCollection*. This method has a parameter to select the layout and other for the initial collection size.

listing 4.8 c and d) the read of the *Particle* data is now performed using the *get* and *set* to read and write the values.

listing 4.8 e) access *gCollection* using the *GasPar* iterators instead of the traditional loop.

At this step, the program already performs the complete simulation using *gCollection*s. In this case, the code modification involves multiple changes since the original code uses a simple array to save particles. If the original code relied on the *List* interface (e.g., as in the JECoLi case study) and accesses to data use the *get* and *set*, the modifications would be restricted to the creation step (listing 4.8b).

The first evaluation compares the original implementation with the base *GasPar* implementation (figure 4.21a). The evaluation measures the runtime of the *runiters* method that runs the simulation (which

```
public Particle one[];
```

```
public gCollection<Particle> one;
```

a) declare collection.

```
one = new Particle[mdsize];
```

```
FactorygCollectionParticle factory = new FactorygCollectionParticle();
one= factory.creategCollection(sVersion, mdsize);
```

b) initialise collection.

```
positionx = one[i].xposition;
```

```
positionx = one.get(i).getPositionX();
```

c) read position x coordinate.

```
one[i].xposition = positionx;
```

```
one.get(i).setPositionX(positionx);
```

d) write position x coordinate.

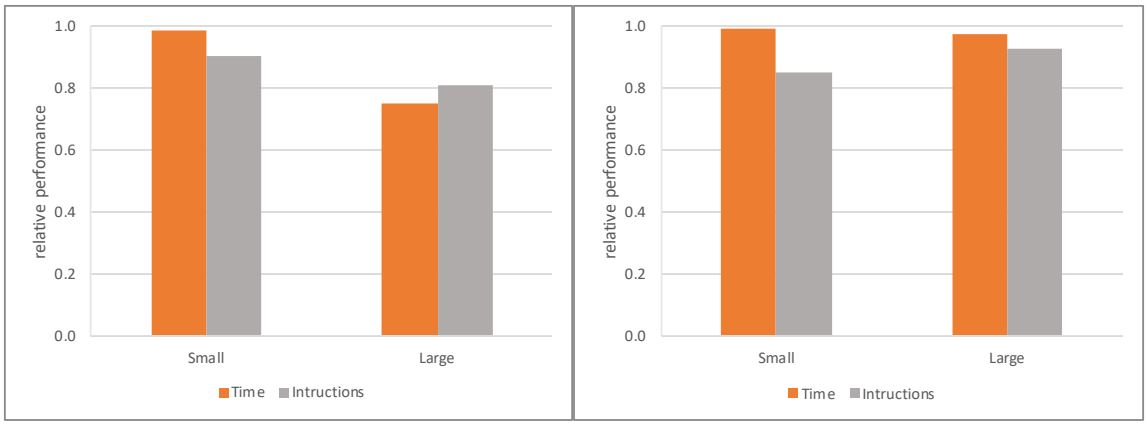
```
for (i = 0; i < mdsize; i++) {
  one[i].force(..., i, this.one);
```

```
for (gIterator it = one.begin(); it.isless(one.end()); it.inc()) {
  force(..., it.get(), c2);
```

e) call the *force* method

Listing 4.8: Code implementation: Original vs *GasPar* Collections

excludes the time for creation and initialisation of the particles). The performance drops to ∼0.75 for the large size due to an increase in the number of instructions. The original version uses a traditional for loop, and data accesses are index accesses. Our approach uses *GasPar* iterators to process the collection. These allow optimisations to be easily applied in the future but adds some overhead to the base implementation in this case study (see section 4.2).



(a) AoP layout (b) SoA layout

Figure 4.21: Approach overhead

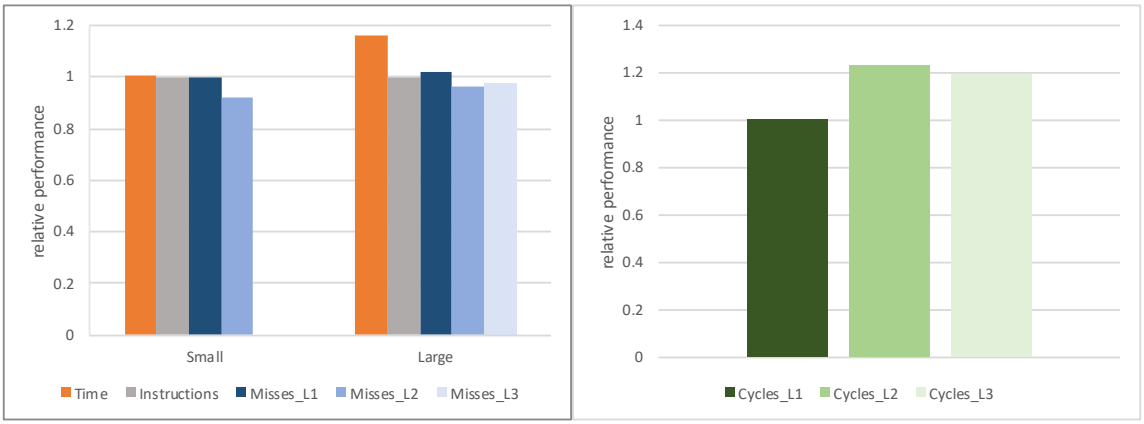
In order to use the SoA layout in the original JGF several modifications are required: transform the *Particle* collection, change several method parameters and change the data accesses. The *Particle* collection tranformation implies changing the *Particle* into a class *Particles*, which represents all particles

in the system. For this purpose, the evaluation transforms the *Particle* fields into arrays. The methods that received a *Particle* now receive the *Particles* and the index that indicates the particle position in the array. Data access now use *one.X[i]* instead of *one[i].X*. These changes make the code more complex. The transformation to *GasPar* is simpler, and enable both layouts.

Figure 4.21b shows that the performance of these two versions is equivalent when using the SoA layout. The performance of the approach is ∼0.97 of the manually converted JGF version. In this case, the compiler was able to remove most of the iterator overhead.

The AoP layout makes it possible to test the sorting optimisation: organising objects in memory by their access order. It is performed by simply calling the method available in the *gCollection* interface. Figure 4.22a shows the improvements by applying object sorting in the collection before executing the simulation. Unsurprisingly, the sorting optimisation only has an impact on large problem size. This optimisation does not improve the small size since the problem fits in the last level memory cache.

The algorithm uses the quick-sort algorithm [Hoa62] for sorting the objects, which has an average complexity of $O(n \log n)$, which is lower than the complexity of the problem ($O(n^2)$). Therefore, the bigger the problem, the less will be the sorting algorithm overhead impact.



(a) Relative performance

(b) Improvements in access data

Figure 4.22: Sorting optimisation improvements

Figure 4.22a shows that the sorting improves the performance but does not explain why. Overall the data access should be more efficient, but the figure 4.22a shows the opposite (the number of misses increases). We decided to obtain additional performance counters to clarify this point. The processor has several counters that count the cycles wasted due to cache misses[12]. Figure 4.22b shows that the sorting

[12]CYCLE_ACTIVITY:CYCLES_L1D_PENDING, CYCLE_ACTIVITY:CYCLES_L2_PENDING, CYCLE_ACTIVITY: CYCLES_LDM_PENDING.

optimisation decreases the total cycles spent waiting for the data. Thus, the performance improvement is due to more efficient data access.

The second test compares the two layouts available in *GasPar*. The test can change the AoP layout to an SoA by changing only a program parameter. Figure 4.23 shows the relative performance by using the SoA layout, which improves performance by ∼2.6 times for the large size. In the small size, the optimisation only improves 1.2 times since the problem fits in the cache, making layout impact lower. The cache misses decreases due to memory footprint decrease and spatial locality improvement. The table 4.11 shows the footprint in memory for each layout. A *Particle* in the AoP layout takes 96 Bytes and in the SoA layout takes 72 Bytes.

In this evaluation, there is also an improvement in the number of instructions due to a change in the layout. However, the impact on the executed instructions is small since the total of instructions that use collections is around 15% of the total number of instructions.

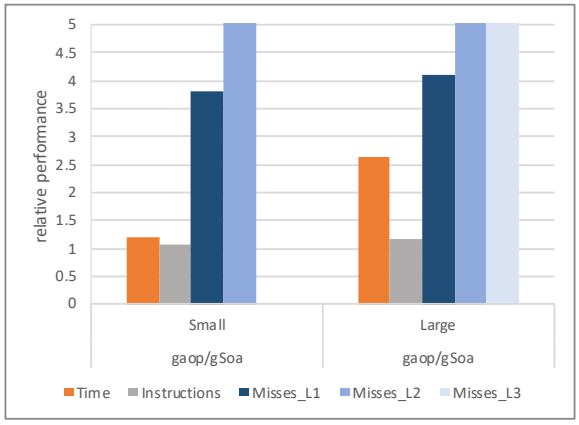


Figure 4.23: *GasPar* collections AoP vs SoA[13]

	Particle	Small	Large
AoP	96 Bytes	824 KiB	79 MiB
SoA	72 Bytes	618 KiB	59 MiB

Table 4.11: Problems sizes

[13]The graph are truncated at five units since the improvements are huge (in the large size improvements on L2 are 70x and on L3 are 3400x, in the small size improvements are 100x).

4.4.2 Tiling optimisation

The tiling optimisation optimises a single program step, which is different from the previous optimisation (has a performance impact on all the program). For this reason, we will first analyse the program execution profile. Figure 4.24 shows the execution profile[14] which indicates that the *force* method takes most of the time (99.9% of the time).

Figure 4.24: MD execution profile

The main focus is to optimise the *force* method by using the *gSplitMapJoin* mechanism to apply the tiling optimisation over this method. The method receives two collections: the elements in the first collection interact with the elements in the second collection. The outer loop controls the first collection accesses, and the second collection is accessed by the inner loop. The tiling optimisation can be applied by dividing the second collection. The listing 4.9 shows the annotation that generates a new method that applies the tiling optimisation. In this case, the second collection is divided into several parts, and the original method is called on each part.

```
@gSplitMapJoin( name ="Tile", map ="map",
              split ={"...","none","Virtual"},  join ={"...","none","default")
public static void force(..., gCollection<ParticleLine> p1, gCollection<ParticleLine> p2) {
  ...
}
```

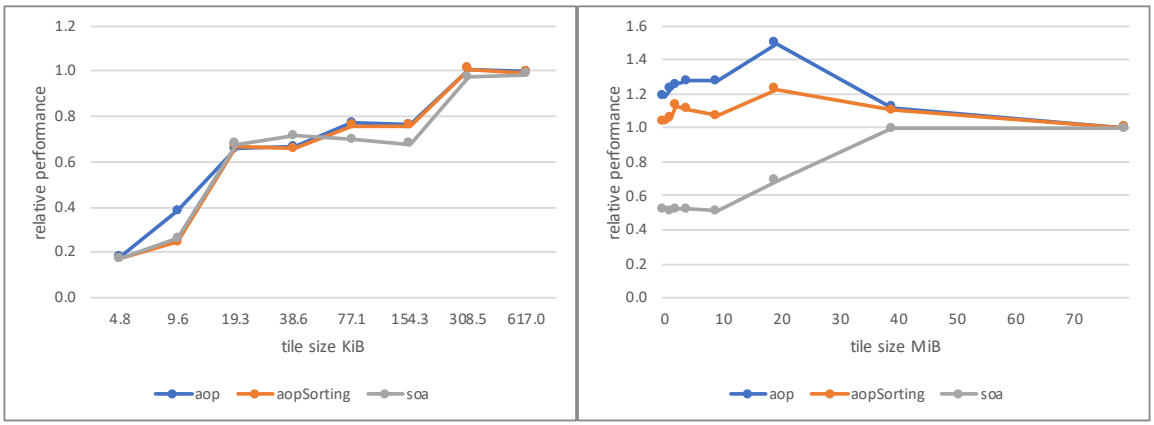
Listing 4.9: Annotation to apply tiling optimisation

The optimisation needs an additional parameter for tuning the number of subdomains (number of tiles). Figure 4.25 shows the impact of the tile size on the performance. The values compare the performance with the versions without tiling. The optimisation improves the performance in $\sim$1.5 for the *aop* with 20 MiB tile size. This impact makes *aop* faster than *aopSorting* due to *aopSorting* needs to sort the collection: more instructions and more data accesses.

[14]The execution profile uses the VisualVM tool in sampling mode.

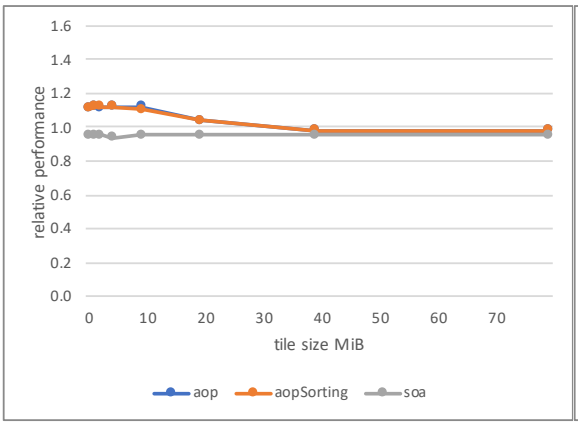
The number of instructions in both *aop* and *aopSorting* reduces in more than 10% when tiling is applied (figure 4.25c). This result is not expected and can be explained by more aggressive compiler optimisations since the inner loop becomes smaller and is executed more times.

The performance of the SoA layout does not improve with the tiling optimisation. As we can see for this layout, the best performance is obtained with only one block, which has the lower instruction overhead. The SoA layout is still the most efficient version (it executes faster than any other).

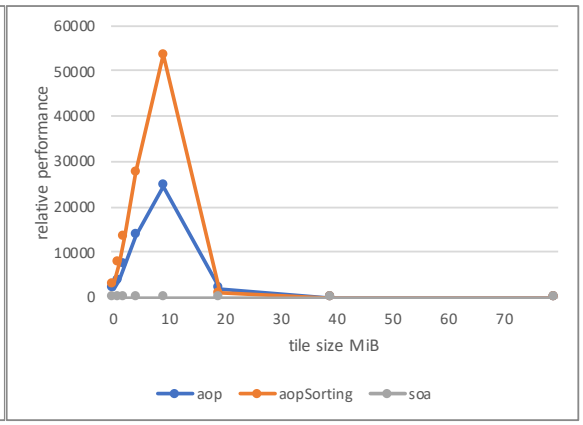


(a) time (Small)

(b) time (Large)



(c) instructions (Large)



(d) misses in L3 (Large)

Figure 4.25: Tiling optimisation impact

In the small size, the optimisation decreases the performance due to the data remains in cache L3 (figure 4.25a). Thus, the tiled data access does not significantly reduce the number of cycles, and the technique needs more instructions to solve the problem (figure 4.25c).

In both sizes, for a single block (equivalent to process the problem without tiling), the performance

is ∼0.99. It shows that the approach introduces little overhead to apply this optimisation. The best improvement is for a block size of 20 MiB in the *aop* (see figure 4.25b). If we analyse the miss rate in the original versions (table 4.12), this result is expected since the *aop* has the highest miss rate. As for the small size, it already has a low miss rate.

	L2 misse rate for Small Size	**L3 miss rate for Large Size**
aop	0.037	**0.451**
aopSorting	0.040	**0.447**
soa	0.014	0.001

Table 4.12: Miss rate analysis

4.4.3 Parallel execution

The MD allows performing force calculation in parallel. However, the JGF implementation uses Newton's third law, which reduces the number of the calculations but introduces concurrency in data access in parallel execution. Different threads access the same data concurrently(to write the forces), which can be avoided by a concurrency control mechanism, where only one thread has access to the data at the same time, or create a new temporarily structure where the thread stores its values. The second alternative needs to reduce the values at the end of the force step. The evaluation uses the second alternative since the approach provides a mechanism to support this alternative, and the approach implements this alternative in a module that can enable or disabled. For the primitive variables (such as *epot*), the evaluation uses the alternative control concurrency mechanisms. In this case, the use of exclusive data accesses mechanisms has a small performance impact since the variables are only rewritten once for each particle (i.e., in the outer loop).

The developed parallelism module uses the Annotation tool. The listing 4.10 shows the annotation that is used to generate the code for parallel execution. The evaluation uses three types of scheduling[15]: *Block*, *BlockBalance* and *Dynamic*. The annotation implementation breaks the outer loop body into several parts and processes these parts in parallel. The *P2* collection needs private force data to avoid the data races. At the end of processing, the approach calls to reduce *md::joinPrivate*.

The UML tool creates the *PrivateForce* method. The developer defines which fields should be thread private. On the other hand, the developer has to define the *md::joinPrivate* (listing 4.11). The *md::joinPrivate* joins the data from a private collection to the original. In this case, the values are cumulative, so the reduce uses a sum operation.

[15]Replace X by *Block*, *BlockBalance* and *Dynamic* (for more details see section 3.2.2 Parallelism).

```
@gSplitMapJoin( name = "X", map = "mapParallelX",
             split   = {"...","Virtual", "PrivateForce"},
             join    = {"...","default","md::joinPrivate")
public static void force(..., gCollection<ParticleLine> p1, gCollection<ParticleLine> p2) {
          ...
}
```

Listing 4.10: Parallelism annotation

```
public static void joinPrivate(Particle col, Particle ret){
    ret.setForceX(col.getForceX() + ret.getForceX());
    ret.setForceY(col.getForceY() + ret.getForceY());
    ret.setForceZ(col.getForceZ() + ret.getForceZ());
}
```

Listing 4.11: Method defined by the developer to reduce private data

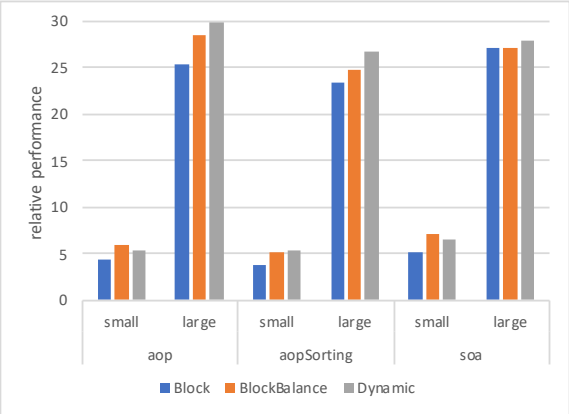


Figure 4.26: Scheduling impact

Figure 4.26 shows the parallel execution improvements[16]. For the small size, the speed-up is limited to ∼7. The best result uses the SoA layout with *BlockBalance* scheduling. This problem is small when compared to the cost of parallel execution (the execution takes 2.24s). The execution profile (figure 4.27 analyses a main thread in the *moldyn.md.one_iteration*. The execution uses only one thread to know how much work would be done in parallel. The domain decomposition (*split* method) used takes 296ms time that includes the creation of the new structures. There is also a reduction method that takes only 98ms. The *moldyn.md.cicle_mkekin* method is not executed in parallel. In this case, the evaluation executed in parallel ∼95% the total workload (*CountDownLatch.await()* represents the main thread waiting for the other threads to calculate the problem). Therefore, considering that we execute the code in 16 threads, the theoretical maximal speed-up is ∼9 times.

[16]Compares with the same sequential version and use the number of threads that obtain the best performance.

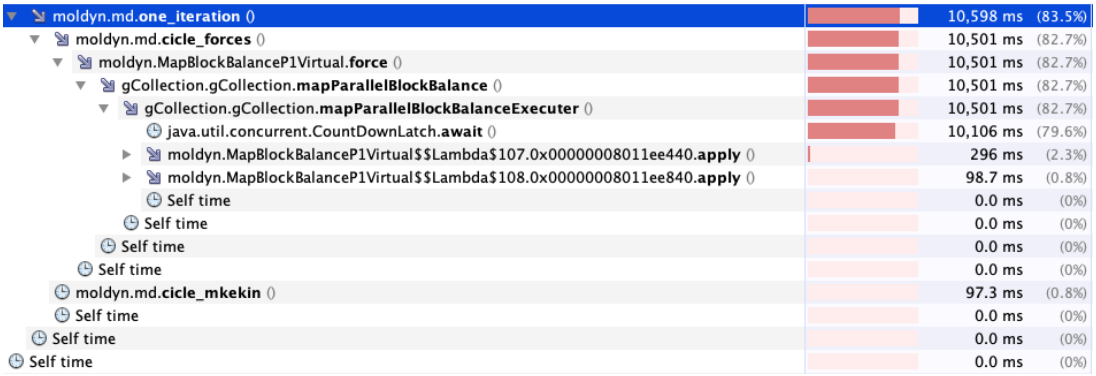


Figure 4.27: Execution profiler for *BlockBalance*

In the large size, the execution time reduces ∼30 times. The layout AoP shows the higher scalability. The other two layouts improve the performance at ∼27 times. In this case, the limitation is imposed by the hardware. The hardware has two processors, where each processor has 12 cores (24 virtual cores with *Hyper-Threading* technology). Intel[17] claims, the virtual cores can bring up to 30% improvement, which means a limit of ∼31 times. In short, the approach gets close to the limit defined by the hardware.

4.4.4 Composing optimisations

Until here, all evaluations only use one optimisation at once. In this section, the evaluation tests the composition of optimisations. In the approach, the optimisations are modules that can be enabled or disabled in the program execution. In *GasPar*, composing optimisations is simple: only include two optimisation modules in execution that the developer validates. The evaluation uses two new versions: *tileP1+tile* and *dynamic+tile*. *tileP1+tile* adds a new level of tiling that also performs a partition of the outer loop in the original version. *dynamic+tile* uses the best parallel version and adds the tiling to the inner loop.

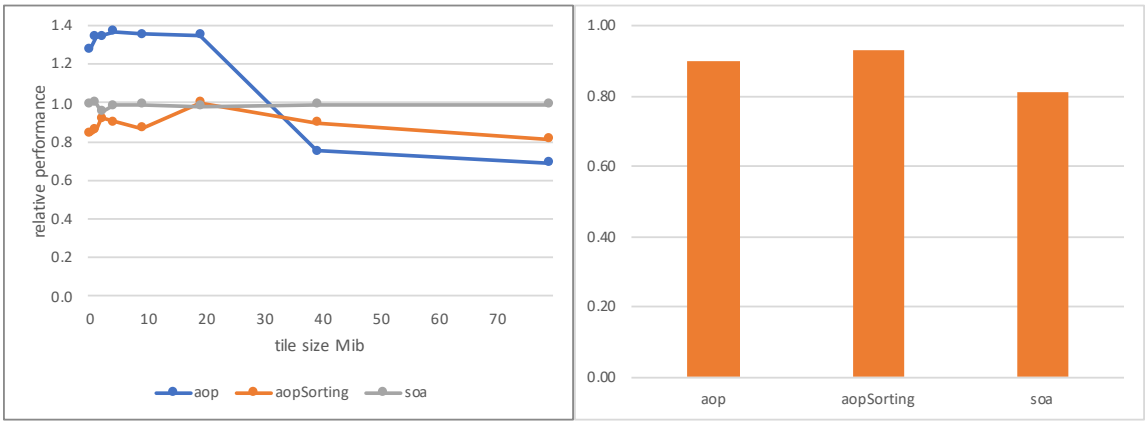
To apply the modules, the developer modifies the external optimisation module (*tileP1* or *dynamic*) and change the original method call to call the internal module (*tile*). The listing 4.12 shows the differences in the external module(*dynamic*) to use the tiling optimisation (*tile*).

The two tiling levels improves performance in the *aop* version ∼1.4 times (figure 4.28a). For the others, there is no improvement in the performance. *aopSorting* improves the locality by organising the elements in the memory by the order they are accessed. The tiling also changes the order in which the elements are accessed, thus, the optimisation has an unpredictable impact. *soa* optimisation has no

[17] `https://software.intel.com/en-us/articles/how-to-determine-the-effectiveness-of-hyper-threading-technology-with-an-application`

```
public class Dynamic ...
{
  (...)
  public static void force( gCollection.Parameters parameters){
        md.force(...);                    Tile.force( ...);
  }
  (...)
}
```

Listing 4.12: Code example of composition of optimisations



(a) *tileP1+tile* (b) *dynamic+tile*

Figure 4.28: Optimisations compositions benchmark (Large size)

improvements since the version is already efficient in accessing the data.

Figure 4.28b shows that the composition *dynamic+tile* reduces the performance. Tiling improves access to data that reduces the wasted cycles in the processor. *Hyper-Threading* technology allows the execution of two threads simultaneously by using the wasted cycles on a second thread. Thus, the *Hyper-Threading* gains reduce when those wasted cycles decrease. In conclusion, improving data access lowers the gains of using two threads in the same core. The performance decreases since the optimisation causes additional overhead.

4.4.5 Complex entity - API closer to the domain

The previous evaluation uses the same API as the original JGF code. A particle has direct access to nine values that represent the 3D position, velocity and force. In the JGF the concept of 3D coordinates disappears in the code. The next evaluation analyses the execution cost to introduce this 3D coordinates abstraction.

The figure 4.29 shows that the new API/Layout (*gaop3D*) has a high cost with the AoP layouts. The performance decreases to ∼60% in the small size and ∼25% in the large due to the memory footprint (more noticed when the problem increases) and the need to perform one more load instruction to access the data. The table 4.13 compares the data footprint in memory. For instance, in the large size, the problem increases to 185MiB in memory instead of 79 MiB. The figure 4.30 represents both layouts in memory. In the JGF, there is an array of pointers that refers to one object that contains all particle data. In the new API, the object has pointers to the other three objects: position, velocity and force. Each object stores three doubles that represent the point in the Cartesian coordinate system. The footprint in memory increases by using the pointers, the Java headers and the memory alignment.

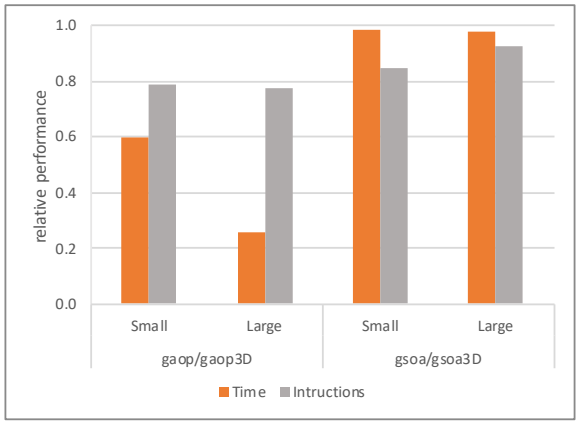


Figure 4.29: Performance using a model of entities closer to domain

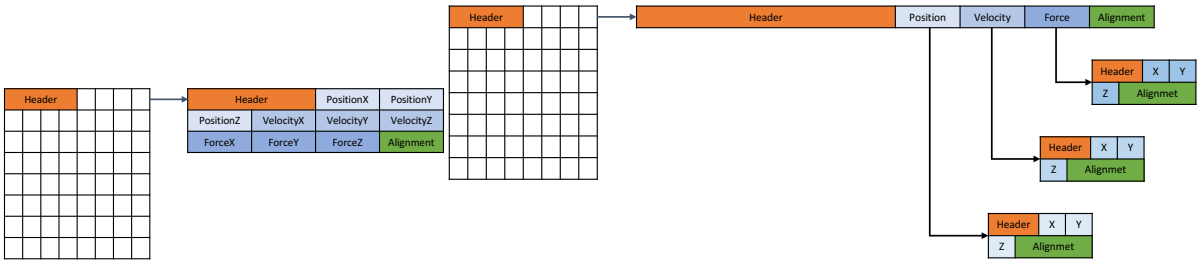
	Particle	Small	Large
AoP layout	96 Bytes	824 KiB	79 MiB
AoP3D layout	224 Bytes	1922 KiB	185 MiB

Table 4.13: Memory footprint from all layouts

The number of instructions increases due to the need to perform one more load to access the data (listing 4.13). To access to the coordinate x from position it needs: access to the particle; access the vector position and finally access the x (listing 4.13 blue colour). The black colour (listing 4.13) is the example in the original version that requires one less instruction.

The performance is equivalent in the SoA layout since the data footprint in memory is the same. There are more instructions due to the need to create an auxiliary structure for simulating the coordinate system[18].

[18]The compiler can remove the auxiliary structures by Escape Analysis, but the optimisation applies only after a few iterations.



(a) AoP layout (b) AoP3D layout

Figure 4.30: Comparative APIs layout

```
load &particles, %eax
load 24(%eax), %ebx
```

```
load &particles, %eax
load 24(%eax), %edx
load 24(%edx), %ebx
```

Listing 4.13: Pseudo assembler

4.4.6 Summary

In this evaluation, we test the *gSplitMapJoin* mechanism. This mechanism allows introducing the tiling optimisation in a simple form and separation of the domain code from the optimisation code. The optimisation improves the execution in ∼1.5 times. Yet, the mechanism makes it possible to implement parallel execution. The private thread data is used only to define the private thread fields and reduction method. Finally, it was evaluated an API closer to the domain that introduces costs in performance, but with the most efficient layout (SoA), the performance cost is negligible. The approach allows improving the performance in ∼ 4 times with the optimisation, and in ∼93 times by enabling the parallel execution.

4.5 Extending collection - Matrix multiplication

Matrix multiplication (MM) is one of the most widely used routines and whose implementation well studied. MM generates a new matrix where the element in the position i, j is obtained by the multiplication of row i of matrix A with the column j of matrix B. For the operation to be possible, the number of columns in matrix A has to be equal to the number of rows in matrix B. A naive implementation has low performance, which can be improved using an optimised MM kernel, introducing data tiling techniques and executing in parallel.

A naive MM implementation is based on the dot product of two vectors: the element Cij of the result matrix is computed from the dot product of line i from matrix A with the j column from matrix B (figure

4.31a). An optimised kernel computes several matrix elements C in a single loop (figure 4.31b). The kernel implements the tiling optimisation for registers, where a (mini-)column of A is brought into registers and multiplied with a (mini-)row of B to compute a (small) block of the C matrix that fits into registers. This optimisation needs to handwrite the loop unrolling (code to compute the small block of C).

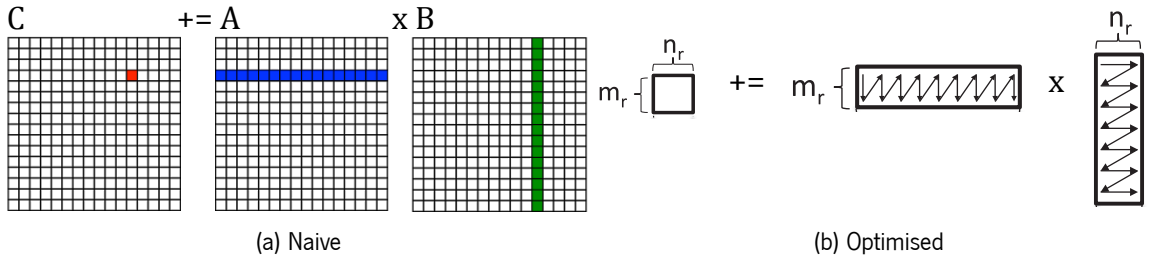


Figure 4.31: MM Traditional kernel

This evaluation uses an optimised matrix multiplication kernel. The kernel processes 4x2 elements in each loop body, enabling the variables to stay in the register, reducing the number of memory accesses (the load and store instructions reduces). In the base version, the loop body calculates a single matrix C element that results from adding to this element the result from multiplication of one matrix A element with one matrix B element (in a way similar to daxpy). The optimised kernels apply the tiling optimisation to a small block so that some variables remain in registers. For this, it needs to apply the unrolling optimisation in the loop body. This evaluation uses the order *kij* for loops, a tiling decomposes the loop i where size is 2, and loop k with size 4. With this setting, the elements of matrix B are reused, and the matrix A elements stay in registered in the outermost loop.

Traditionally, tiling allows sub-matrices to stay in the cache for a longer time. Additionally, the packing optimisation can improve performance since the data footprint used for processing is better (improve spatial locality).

One (or more) loops can be executed in parallel, which is equivalent to the computation of submatrices in parallel. The most common strategy is to assign different matrix C blocks to multiple threads since each thread writes to one different submatrix C (there is no concurrency in data writes).

This evaluation explores two questions about the approach: the first is the approach extension to support data with several dimensions (e.g., matrices); the second studies the packing optimisations which are common in MM implementations. The tiling in the matrix multiplication originates accesses to the submatrix data that will remain in the cache. Packing can group this data into small matrices to improve cache usage.

The evaluation extends the *GasPar* to support matrix-like data structures[19]. The new API enables it to iterate over the matrix by columns and by rows. For this purpose, the API implements two new methods that create different iterators: one accesses the matrix by columns and another accesses by rows. It also allows index access to an element with the *get* and *set*. Figure 4.32 shows the new classes that support this new matrix API. When the developer iterates by rows, the developer uses *beginRow()*. This method returns an instance of the *gIteratorRowDouble* class [20], that represents a matrix row, providing access to the elements of a matrix row. Using this class, the developer iterates over the collection as in the usual method (e.g., calling begin, etc).

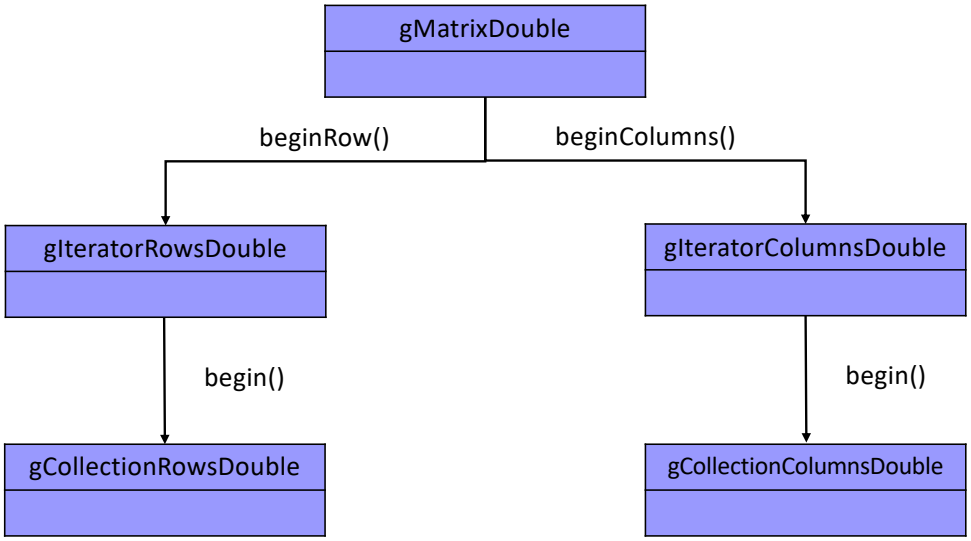


Figure 4.32: Matrix API[21]

A question behind this evaluation is the matrix layout. In the previous case studies, the SoA layout always provided the best performance. However, a new question arises regarding the matrix layout: using an array of pointers for arrays or using a single vector (figure 4.33). Java uses the AoA layout. AoA layout has an array of pointers to the matrix lines, and a line is an array, so each line access is done by a load. Vector layout uses a unique array (contiguous space in memory) where the data access uses a calculation. In Java, this calculation is specified by the developer. In other languages, such as C, it is possible to implement this layout using traditional access. AoA needs additional space to store the pointer array. The *GasPar* approach allows these options to be transparent for the developer (i.e., use the same

[19]The tool does not support the matrix API since it is a prototype for the matrix access, developed for evaluation purposes. The code to support the new API was handwritten.

[20]Is a *gIterator* and *gCollection* at same time.

[21]It shows the concrete classes to empathise the new implementation.

matrix API for both layouts). This evaluation starts by comparing the performance of both layouts. The first layout decreases the number of calculations, but it also introduces new memory accesses. Table 4.14 shows memory footprint is need for each of the sizes tested.

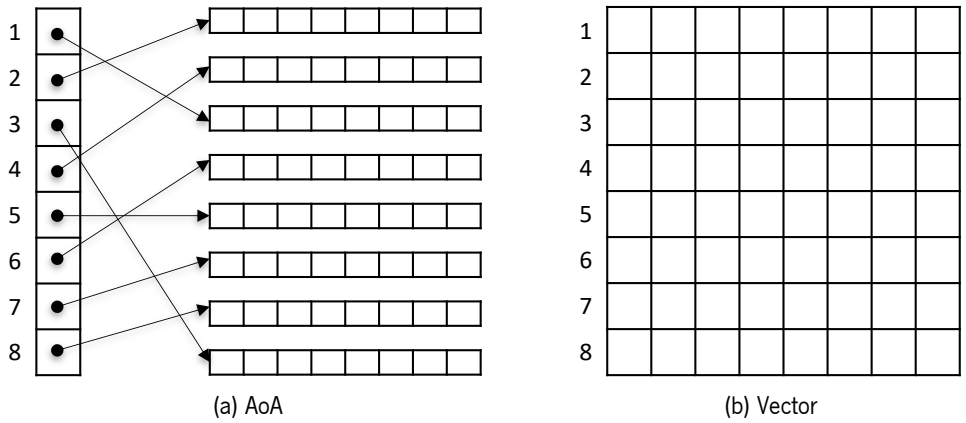


Figure 4.33: *aoa* vs *vector* layout

	Problem size (MiB)	Line Size (KiB)
1024	192	64
4096	3072	256
16384	49152	1024

Table 4.14: Problem size

Figure 4.34 shows the impact on the performance. The approach has a maximum cost of $\sim$0.88 over traditional code development. The *GasPar* API allowed writing the low level kernel using iterators (note: the kernel contains register tiling and loop unrolling in the code), but it introduces overheads. Despite performance loss, the use of iterators allows the application of the remaining optimisations without modifying the domain code.

The *vector* causes a substantial loss of performance, and the best case can not reach 0.60 of the native implementation performance. For this reason, the vector option was discarded in the remaining tests.

4.5.1 Tiling optimisation and parallel execution

In this evaluation, all matrix sizes are larger than the cache memory size. Therefore, it needs to access the main memory. As already defined above, the kernel uses the *kij* ordering for loops which implies that the innermost loop needs a row of matrix *C* and a row of matrix *B* (figure 4.35). The middle loop (loop i)

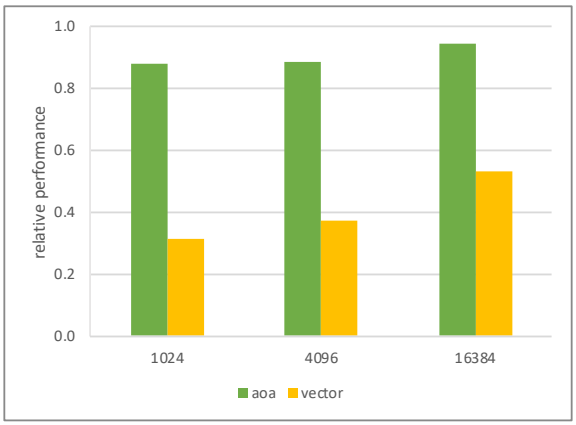


Figure 4.34: Layouts performance

uses the same row of matrix *B* and loads a new line of *C* and carries a value of *A*. To take advantage of the data locality, it needs to ensure that the multiple lines of matrix *B* remain in the cache.

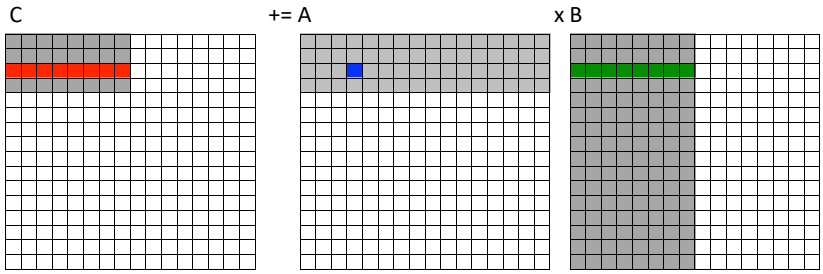
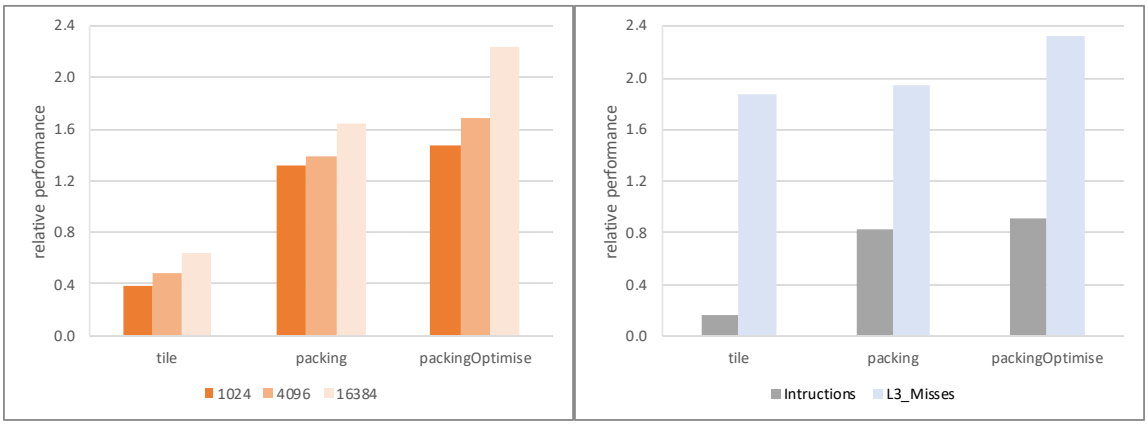


Figure 4.35: Kernel elements + tiling

The *gSpliMapJoin* mechanism decomposes the problem into subproblems. The subproblems must remain in cache memory to take full advantage of data locality. The matrix *C* subproblem is defined with the size of 32 rows by 512 columns (tile example in figure 4.35 is 4x8). Thus, the matrix *B* line, for this subproblem, has 512 elements. Additionally, the submatrix *C* has 32 lines to stay in the L2 cache while calculating the subproblem. For matrix *A*, it needs to load new values into registers in the external loop (loop *k*).

The evaluation compares three tiling options. The *tile* option uses virtual views that use collections where the *begin* returns an iterator to the starting subcollection position. The *packing* option divides the matrix with *splitondemand*, where the data reads trigger the packing of the submatrix before processing the subproblem. After processing, it writes the packed submatrix into the original matrix. The *packingOptimise* option creates all packed matrices for matrices *A* and *B* at the beginning and uses the split on demand on matrix *C*.

Figure 4.36a compares the three tiling optimisations impact. The best version is *packingOptimise*.



(a) time (b) instructions+l3_misses[22]

Figure 4.36: Tiling and packings optimisations impact

This version is better than the *packing* since it prevents the data of matrices *A* and *B* from being reloaded several times to the packing matrix. Each submatrix of *C* is loaded only once, so the *splitondemand* uses less space in memory.

The tiling optimisation improves data accesses by decreasing the number of misses, as the figure 4.36b shows. This improvement has a positive impact on runtime for two versions (figure 4.36a): *packing* and *packingOptimise*. However, for the *tile* option, the runtime increases due to an increase in the number of instructions (figure 4.36b). In this case, the virtual collections introduce additional instructions since they disable some compiler optimisations (e.g., bounds check in collection accesses).

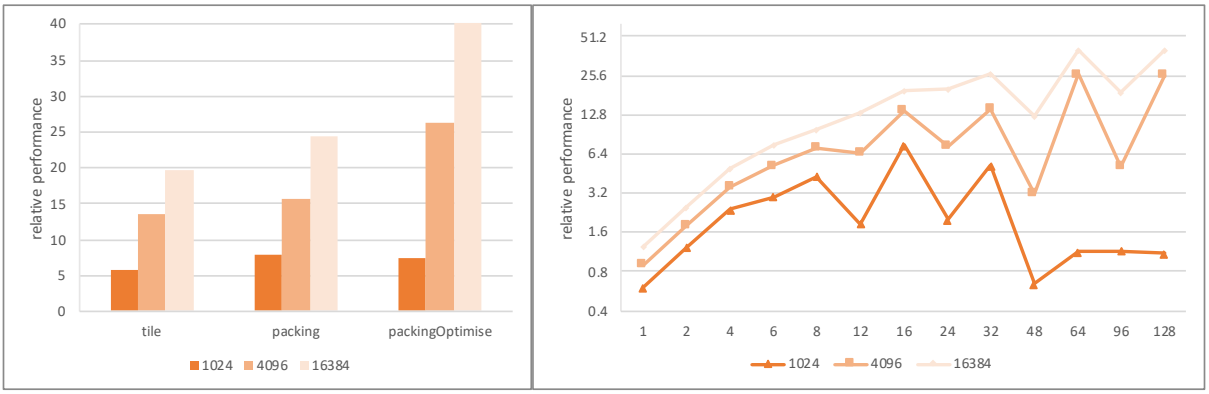
There are many other possible tiling combinations. The ones presented in the figure 4.36 provide the best (and more representative) results. There was a preliminary study not presented in this document since it is not relevant to this evaluation.

The parallel execution uses the block-scheduling strategy where the subproblems are organised into blocks[23]. The parallel execution requires a mechanism to control the concurrency in packing since the *C* matrix uses packingondemand. To solve the problem, the evaluation utilises the strategy present in the *gCollection* (create a private matrix for each thread).

The figure 4.37a shows that the parallel execution decreases the execution time. The *tile* option has the worse improvement in the runtime, while *packingOptimise* shows the best improvement. These results reflect the sequential version performance (see figure 4.36a). The parallel execution and packing enable a 40 times improvement for the larger size: ∼2.3 times from the data *tile/packingOptimise* optimisation

[22]The columns show the average of all three matrix sizes.
[23]One block has one or many subproblems.

and the remainder from the parallel execution.



(a) Parallel+optimisations[24] (b) *Parallel+packingOptimise*

Figure 4.37: Parallel execution impact

The performance for all versions (*tile*, *packing* and *packingOptimise*), as the number of threads increases, follows a similar trend-line (see in appendix figure 7.1). The figure 4.37b shows the trend-line of the faster option. There is a performance drop when the number of threads is not a power of 2 due to load unbalancing. In that case, there is a thread that processes more elements than other threads, so the execution time reflects this thread time. Although the machine supports only 48 threads, at the same time, there is an improvement with 64 and 128 due to using all threads and better load balance. The lower speed-up occurs in the small size since parallel execution cost is proportionally larger (leading to higher parallelism overhead).

4.5.2 Libraries for matrix multiplication

Using *GasPar* enabled a performance improvement when compared to the previously developed base kernel. For this, the best performance is obtained with packing optimisation and parallel execution. The OjAlgo library developers analysed the performance of some libraries available for matrix multiplication for Java. The study (results shown in figure 4.38) are based for the JMatBench on 2018-04-04 [25]. The most efficient libraries are the EJML for small sizes and ojAlgo for larger matrices. For this reason, the evaluation compares *Gaspar* against these three libraries.

Efficient Java Matrix Library (EJML) is a linear algebra library for manipulating real/complex/dense/sparse matrices. The library was designed to be as computationally and memory-efficient as possible for

[24] The gains are relative to the *aoa* version and for the number of threads that have minimum execution time.
[25] `https://github.com/lessthanoptimal/Java-Matrix-Benchmark`

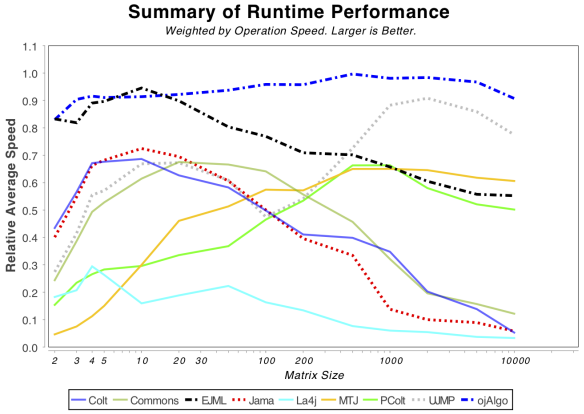


Figure 4.38: JMatBench: Summary Benchmark performance

small and large matrix, and accessible to both novices and expert developers. These goals are accomplished by dynamically selecting the best algorithms to use at runtime, clean API, and multiple interfaces. EJML is free, written in 100% Java and has been released under an Apache v2.0 license. EJML uses three kernels to calculate the matrix multiplication. When matrix B has only one column, it uses the first kernel. The second kernel is for small matrices, i.e., when the number of columns is less than 15. This kernel also uses the *ijk* loops order. The third kernel is used for all other cases, and the order of the loops is *ikj*. All kernels are sequential and do not use tiling or packing optimisation.

ojAlgo[26] is a linear algebra library also developed in pure Java. In figure 4.38, ojAlgo presents the best libraries performance in most problems sizes. Unlike the other libraries, ojAlgo uses domain decomposition to optimise code as well as parallel processing. The library begins by decomposing the problem into subproblems that execute in a thread pool.

Additionally, the evaluation includes jBlas in this test. The jBlas uses the blas and lapack through JNI calls to the corresponding methods from the Fortran library. Thus the library is not developed in Pure Java, which makes it dependent on the execution platform.

The table 4.15 show the library versions used in this evaluation. The figure 4.39 shows the relative library performance compared to the *GasPar* base. The *optimised* version is *packingOptimised* with sequential execution. The *parallel* use same version with parallel execution.

As expected, EJML is the lowest-performing library since it does not use the major optimisations (figure 4.39). ojAlgo performs better for the small size. In this size, the matrix fits in the cache, and the bottleneck is not memory since the innermost loop is loop *j*. Additionally, ojAlgo uses parallel processing that makes it faster than our *optimised* base. When compared to the *parallel*, the performance is similar in the smaller

[26] `https://www.ojalgo.org/`

Library	Version
EJML	0.38
ojAlgo	47.3.1
jBlas	1.2.4

Table 4.15: Libraries versions

size. jBlas uses a sequential version of ATLAS. The jBlas has better performance than base *GasPar*, but the *optimised* is better than jBlas. Using parallel processing reduces the execution time by nearly 46 times for the larger size.

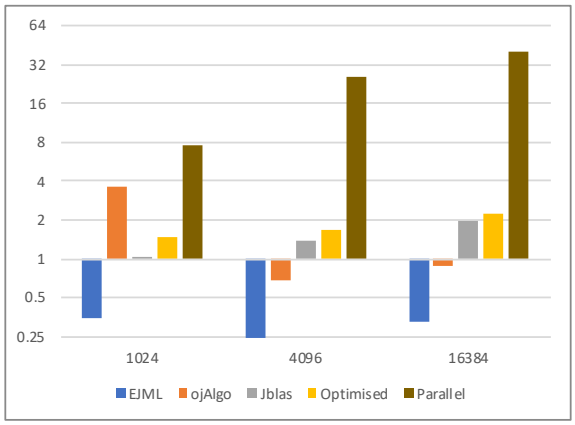


Figure 4.39: Comparation *GasPar* with other Libs

4.5.3 Summary

This case study allowed us to extend the approach to support matrices. The code developed is a prototype that shows the approach potentialities to accommodate new containers. The new API allowed us to use a complex kernel.

The virtual collections have worse performance, inhibiting the application some optimisation by the compiler. The packaging optimisation eliminates this limitation and improves performance.

Finally, *GasPar* enables parallel execution to reduce the execution time by more than ∼20X in the larger size.

4.6 Conclusions

The approach allows the use of the most efficient layout without neglecting the programmability. The evaluations use the same API as the Java collections and additionally improve the efficiency in accessing data through the layout. Additionally, the approach allows the application of the most common optimisations, hiding implementation details. All optimisations available in the *GasPar* are pluggable (can be removed or introduced easily by the developer).

All evaluations had an improvement in performance. The figure 4.40 shows the improvement in performance in all evaluations. It was possible to reduce the execution time by almost 20 times, for all evaluations that support parallelism. MD reduced the execution time by 50 times. In JECoLi, this improvement is less since the parallel execution is not used. The improvement is due to a more efficient layout. JECoLi's execution time has been reduced by 3 times.

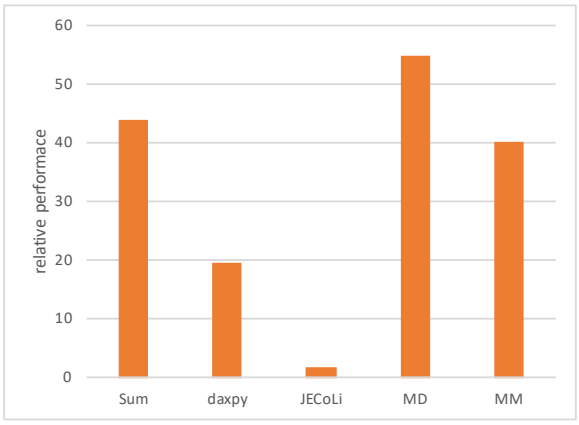


Figure 4.40: Evaluations performance summary

Figure 4.41 shows the layouts improvement in execution time. For Sum the layout improves the performance in ∼4.5 times. In daxpy, the SoA layout has 2.5 more performance. In the MD, the performance doubled with the SoA layout. In MM, the most efficient layout is the base layout, for that reason, there are no improvements.

Java uses Autoboxing and Unboxing to allow the use of its collections with primitive types. However, its use causes a high cost when it is necessary reading and writing the value in the collection. These two operations force the creation of a new object (daxpy).

Our approach allows, starting with a basic AoP version, and to switch to SoA layout. In the MD case, the changes are limited to accessing the data. Manual implementation requires the same changes and additionally needs to redefine the method's parameters. Our approach allowed JECoLi to remain

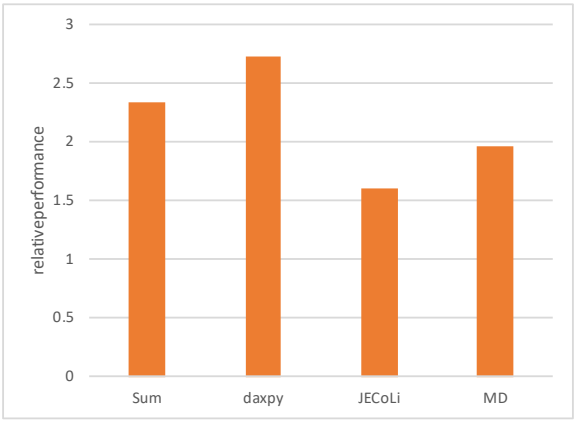


Figure 4.41: Evaluations performance summary (layout improvements)

generic using the most efficient layout. The integration of our approach in JECoLi was simple due to the compatibility with the Java collections.

Our iterators reduce the number of instructions since some instructions are removed when compared with Java iterators. Our approach allowed this reduction by the compiler since collection elements are created all at once and use the same type of data.

The table 4.16 summarises the results that were measured during this work. For each case study, different versions are measured, which are quantified in the #versions column. The #tests column identifies the total number of versions measured with different parameters (e.g., number of threads). The last column gives an estimate of the computation time spent for each case study.

	# versions	# tests	Total time
Sum	37	174	4m x 100 (7h)
daxpy	45	465	5m x 100 (8h)
JECoLi	4	14	1h x 10 (10h)
MD	83	553	233h x 10 (97 days)
MM	18	328	12h x 10 (5 days)

Table 4.16: Evaluations summary (development time)

In Sum and daxpy, several versions were created for comparison (i.e., without using the framework). We replicated this versions by using the framework. It only needs one code for both layouts. At JECoLi, 4 different versions were created. First, we use the original version by the AoP layout for collections. The other versions used the *GasPar* framework, with 3 different layouts: generic types in the layout, AoP layout and the SoA layout. The code is the same for the three versions. In MD, we used two different codes with the framework: the first used the flatten structure, the second used the structure composed of other

structures. These two codes generate sixteen versions with different optimisations. The other nineteen versions did not use the framework and are handwritten. For matrix multiplication, the framework allowed to test two layouts with the same code. The framework enable several optimisations that generated nine versions with the optimisations supported by framework.

Chapter 5

Discussion

The beginning of chapter 3 presented the requirements of the approach. The first two requirements (use domain abstractions in the code, using an OOP approach and deliver high performance) were difficult to provide, at the same time, making them one of the main challenges to address in this dissertation. The use of OOP may introduce an overhead not admissible for HPC. The solution introduced in this dissertation allows the developer to use OOP concepts, such as objects, methods, etc making the code more abstract, and simultaneously, the approach provides support for performance tuning by providing a set of locality optimisations.

Section 2.2 presented the most common locality optimisations. This chapter compares other approaches that support these optimisations with the proposed solution. Table 5.1 summarises the supported optimisations. The first column identifies the optimisations; the second summarises how the approach supports those optimisations, and the last column indicates other frameworks supporting the same optimisations.

	Implementation	Other frameworks
Data Layout	Generation+Data API	JCRNE, Valhalla, Wimmer, SDLT, ASX, Sharma, ASTA
Data sorting	Data API	Hirzel, Chilimbi
Padding and alignment	-	Compiler
Tiling + Packing	Data API + Annotation	OpenACC, Mint, Pluto, Polly
Loop fusion	-	Pluto, Polly
Loop reorder	-	Pluto, Polly
Privatisation	Data API+Generation +Annotation	OpenMP
Parallelism	Data API+Annotation	OpenMP, OpenACC, *Java streams*, Habanero, TBB

Table 5.1: Locality optimisation and parallelism approaches

The next section discusses the programming interface, namely, analyses the iterators options. The discussion in sections 5.2 and 5.3 compares the approaches and their mechanisms to support: data

layout, data sorting, padding and alignment, tiling and packing and finally, parallelism and privatisation. The data layout optimisation has been the target of several research projects. Section 5.2.1 will explain the different kinds of approaches that transform abstracts layouts into more efficient layouts. Section 5.2.2 presents two approaches to implement the sorting optimisation, that use Garbage Collector (GG) to sort the data. Traditionally, padding and alignment (section 5.2.3) changes the data structures definition and data allocation, being simple to implement, or hiding by the library for allocation or by compile-based optimisation. The Tiling (section 5.2.4) is implemented in some approaches by placing annotations in loops (e.g. OpenACC). Other approaches (e.g., Pluto and Polly) rely on a sophisticated analysis of memory accesses patterns. Section 5.3 describes tools that support parallelism, focusing on data parallelism. The final section presents a summary of the approaches advantages and their constraints.

5.1 Programming interface

GasPar supports both Java and STL-like iterators. Moreover, *gCollection*s fully support the Java *List* interface, opening the door to a wider range of applications. This dissertation supports the most used Java container (i.e., ArrayList) [CASL17]. Thus, we can replace this container with a *GasPar* container. For problems where the bottleneck is the access to the collection, the developer can improve the performance by using the *GasPar* collection with the SoA layout, while keeping the domain abstractions.

STL and JCF support the use of collections of polymorphic data. However, they support the polymorphic data only with the AoP layout. The current implementation of *GasPar* shares the same limitation. The section 6.2.1 discuss an approach to implement polymorphic data in *GasPar* with the SoA layout.

The iterators allow the developer to write code that is independent of the collection layout. For example, the developer can use a *LinkedList* instead of *ArrayList*. Moreover, using iterators sometimes has an additional cost since the iterator might disable some compiler optimisations [BVDGN06] or add overhead.

Java iterators provide two methods to process a collection: one tests if there are more elements and the other returns the current element and advances to the next one. The use of this type of iterator has limitations [Wei02]. For example, it is not possible to implement a matrix transpose with these iterators. However, operations over matrices are essential in HPC, so the *GasPar* also extends the iterator API.

GasPar also offers iterators similar to the STL iterator. The STL initialises the iterator with *begin()* and tests if all elements are already processed by comparing the iterator with the last collection element (given by the end method). The next element is performed with an increment to the iterator (e.g., it++). The STL iterator exposes the loop limits that helps to support the parallel execution. In the early versions of the *GasPar*, this iterator simplified the development, now it is used internally. The STL iterator allows the

developer to modify the end of the iteration more easily. This iterator is used to implement the optimisations such as tiling.

GasPar iterators introduce *synch()* method that enables iteration over two or more collections, at the same time, through iterators. STL allows the developer to synchronise iterators, but it is less abstract. In STL, the developer uses *std::advance(it1, std::distance(beginit2, it2)-std::distance(beginit1, it1))* to synchronise *it1* with *it2*. The Boost[1] library developed for C ++ also allows the processing of several collections simultaneously. For this, it provides a zip iterator that abstracts a set of iterators making it possible to access several collections with the same iterator. In the future, this could be an alternative to the approach (e.g., a join iterator).

Robert proposes a new approach to remove matrix indices from the code in linear algebra [BVDGN06] based on a domain-specific language that can be directly mapped into library calls. The development proposed that the processing should occur in different steps. However, that approach is specific for the Linear Algebra domain. In this work, the *GasPar* approach was tested in multiple domains.

The implementation of the proposed iterators, provide a performance similar to lower level programming (e.g., array index-based). It relies on modern JVMs optimisation to eliminate the iterator overhead. At the implementation level, iterators become an integer to control the loop iteration range. In contexts that require the sync among iterators, it becomes important to use the same loop control variable for multiple iterators (as the sync operation does).

The iterators make the code closer to the domain. This thesis analysed different iterators and opted to include both Java and STL iterators in the approach. Java iterators enable to change of a Java collection by a *GasPar* collection. Additionally, the STL-like iterators are well-known to HPC developers.

5.2 Data locality optimisations

5.2.1 Data layout

The choice of the data layout, in traditional approaches, is made at the beginning of the code development. Thus, changing it after the initial coding step implies many changes in the code. On the other hand, the most efficient layouts typically do not use domain concepts, as they are closer to the execution platform. There are several options to allow the choice of the layout in the final development step. The most common options are data encapsulation, code transformation, use of proxies and use of the JVM to manipulate the data accesses.

[1] `https://www.boost.org/`

The data encapsulation technique hides the layout over an API by creating temporary objects (e.g., an adapter). Section 2.2.1 presented one approach that enables multiples layouts with the same API. However, this API is not compatible with the Java Collections APIs. In Java collections, *get* returns a reference to the entity (object), and it can make changes directly on it. In comparison, the JCRNE returns an object copy and forces the developer to set the new object in the collection to update the entity. The proposed approach behaves like Java collections and returns an iterator/proxy that enables direct access to the collection. In practice, the developer writes the same code with our approach as with Java collections. The data encapsulation implementation requires the creation of new objects (e.g., the iterator/proxy) to expose the required API. The overhead of creating this object is removed in most cases, in recent versions of the JVM. However, in older versions of the JVM, the JCRNE has a significant overhead on performance [FSS13], whilst the approach reduces the overhead by using the same object several times (e.g., using the same iterator/proxy to iterate over all the elements in a collection). JCRNE does not support tiling, packing and parallelism. However, it is possible to adapt the proposed implementation to use a JCRNE-based implementation.

Section 4.2 (sum/daxpy cases studies) compared the implementation and performance of this approach against our approach. Listing 5.1 shows the code differences for daxpy: the *setValue* writes the value in the new object; the set writes a new object in the collection Y.

```
for( int i=0; i < collectionX.size(); i++) {\\
   gDouble dy = collectionY.get(i); //returns a reference to an element in the collection (proxy)
   double aux = alpha * collectionX.get(i).getValue() + dy.getValue();
   collectionY.get(i).setValue(aux);
}
```

a) GasPar API

```
for( int i=0; i < collectionX.size(); i++) {
   gDouble dy = collectionY.get(i); //returns a new object, copy all fields
   double aux = alpha * collectionX.get(i).getValue() + dy.getValue();
   dy.setValue(aux); //update all fields in collection
   collectionY.set(dy, i);
}
```

b) JCRE API

Listing 5.1: Java code of different daxpy implementations

The figure 7.2 (in thesis appendix) shows that JCRNE has similar performance that the *GasPar*. The compiler removes the creation of the temporary object (the JiT uses the escape analysis, section 2.3.1, to apply this optimisation), so the generated code is similar to *GasPar*.

In the code transformation approach, the source code is processed and changed to implement the desired layout. The technique is versatile allowing multiple changes to the source code. On one hand,

it is possible to have an approach that allows the developer to specify the code as the developer wishes, which imply the development of a complex tool. On the other hand, it can restrict domain code writing, facilitating tool development.

Sharma [SKK+13] presents a C++ framework that can change the AoS layouts to SoA or hybrid layouts. The tool processes the code changing the layout (source to source approach) specified by metadata. It can also create hybrid layouts (between AoS and SoA) automatically, according to the way that fields are accessed. Our approach could use this mechanism to create hybrid layouts. The tool does not support tiling and parallelism.

Wende [Wen19] suggest one approach based on proxy objects. The approach uses macro-based C++ to generate the code referring to the proxy that allows using both the AoS layout and the SoA layout. This approach is based on a strategy similar to the *GasPar* approach.

Intel suggests using the AoS layout for design and using the SoA layout for performance[2]. SIMD Data Layout Templates (SDLT) is a template library for C++ language that enables abstract code (use of an AoS-based API) and uses the layout SoA in memory. The template library creates an implementation in the pre-compiler step, where the layout is also selected. SDLT uses the C++ operator overloading to enable the traditional access API (e.g., a[k].field1). However, the use of this API introduces new copies of the elements and consequently introduces overhead. To reduce the overhead, SDLT provides an alternative API that implies accessing data using methods (e.g., a[k].field1()). This strategy is similar to the *GasPar* API and implies the same code rewrite (*get* and *set* for each field). However, in SDLT the data structure can only have primitive types, although the *GasPar* approach supports complex structures.

SDLT generates the layouts using the C++ macros. In our approach, the tool generates the layouts. The SDLT supports multiple dimensions (e.g., arrays, matrix...). Additionally, our library also provides other optimisations like parallel execution.

ASTA is a layout proposed by Sung in et [SLH12]. This layout is good for vectorisation since it groups the same field of successive elements. The next memory position stores the next field. This layout uses a hybrid layout to allow vectorisation and to keep the entity data in nearby memory positions. With this layout, the data is ready to be read into vectorial registers avoiding gather and scatter operations. Elements must be grouped taking into account the data size supported by the vectorial instructions in the execution platform. Jubertie [JMF18] propose the same layout. However, the developer uses an approach similar to SDLT to change the layout. There are other approaches similar to SDLT. ASX [Str11] is a library that enables the change of AoS to SoA layout. About SDLT, ASX allows the use of composite structures.

[2] `https://software.intel.com/sites/default/files/managed/01/cd/improving-vectorization-with-intel-simd-data-layout-templates.pdf`

All presented approaches allow the switch from AoS to SoA. *GasPar* allows a similar change but for the Java programming language. Polymorphism is not supported in any of the approaches, which indicates that this feature is not relevant for HPC. The current approach implementation does not support polymorphism, but proxy-based implementations can be extended to accommodate polymorphism. In the future work section, we will discuss this implementation.

The Java language provides an implementation alternative at the JVM level since it can modify the data layout in memory. This approach allows a more efficient layout, but it is only possible to turn AoP into an AoS. Wimmer et. al. [WM08] propose an improvement to the JVM to automatically in-line object fields by placing the parent and children objects in consecutive memory places and by replacing memory accesses by address arithmetic. The authors point out that using arrays as in-lining parents is complex since the Java byte-codes for accessing array elements have no static type information. They claim that an automatic AoP to AoS transformation at JVM level is impossible without a global data flow analysis.

The Valhalla[3] is a new Java project aiming to improve the performance of data accesses, by implementing small objects more efficiently. The project supports in-lining of objects into a parent object (removing the pointer). This implementation removes the identity of the object allowing a more efficient implementation. Thus, a composition of objects becomes a simple structure, removing the headers of in-lined objects. This optimisation can be applied to arrays, implementing an AoS layout, which removes the header and pointers to objects. Moreover, Valhalla will allow collections of primitive types to be more efficient, without the need to use Autoboxing and Unboxing. However, this approach will not provide support for a more drastic restructuring of the layout (e.g., SoA layout). The implementation of this project in Java will facilitate the include AoS layout in our approach.

5.2.2 Data sorting

To sort the data in the memory, we present two alternatives: the first uses the Garbage Collector; the second uses a method that hides the operation of sorting the collection. The data order to obtain the best performance depends on the problem. Chatterjee [CLPT02] presents several layouts for matrices. For example, the Z-Morton layout [TBK06] has the same performance for accesses by row or column. In our approach, we sort objects according to their position in the collection (AoP collection).

Chilimbi [CL99], and Hirzel [Hir07] modified a JVM Garbage Collector to sort objects into memory according to their temporal affinity. Objects that used at the same time are placed in nearby memory zones. The JVM sorts the objects during the garbage copying. This technique still maintains the AoP

[3] `http://wiki.openjdk.java.net/display/valhalla/Valhalla_Goals`

layout and thus cannot avoid the overhead of pointer indirection.

Both approaches propose to optimise the data access to sort the objects in memory through the Garbage Collector. This approach is transparent to the developer, but this technique requires a modified JVM.

In our approach, we order the data in memory using a *sort* method. The objects are reordered in memory according to the index in the collection. This criterion is simple but provides performance improvements in the cases studied. On the other hand, it can be the developer to define other criteria to sort the collection.

5.2.3 Padding and alignment

The proposed approach does not implement this optimisation since this optimisation is well-localised in the code, and it is possible to apply this optimisation in the final development step. Also, compilers/libraries already provide support for optimisations, such as aligned allocation. On the other hand, it is possible to align the data using meaningless padding elements. Padding is also used to prevent false sharing in cache memories. Overall the optimisation techniques change the data structure definition or allocation, which has a small impact on the code. However, the techniques might be useless if the compiler/allocation also performs these optimisations.

In Java, objects are aligned in memory, but they always have a header. So, the array object is aligned in memory, but the first array element is not. The object header forces the initial position to be misaligned in memory. One solution is to ignore the first array positions and the first element of the array to aligned in memory (loop peeling [HCM14]). Without the optimisation, the first elements are not calculated with vectorial instructions. When the application uses the tiling optimisation, the problem is worse since the first element of the tile is not aligned in memory.

Java takes advantage of data aligned in memory since the data address always ends with 0(s). In that cases, it stores the pointer in a compressed way (32 bits instead of 64 bits) by eliminating the 0s from the pointer, thus reducing the space needed to store the addresses in memory.

5.2.4 Tiling and packing

The tiling can use two approaches: loop rewrite or decompose the domain into multiple subdomains. The loop rewrite implies adding new loop(s) in the code and redefine the internal loop(s) limits. It can be implemented manually, by annotating the code or by a specific compiler. In these two last strategies, the tiling optimisation is applied by a code analysis and transformation tool.

The loop rewrite has no meaning at the domain level. However, the domain decomposition technique is abstract (closer to the domain). Additionally, it makes it easy to apply packing optimisation.

OpenACC and Mint [UCB11] are two programming frameworks that provide OpenMP-like directives to support the loop tiling through a specific loop clause. Both approaches apply tiling through primitives which simplifies code development by reducing development errors. However, primitives are handled at compile time which prevents setting the tile size at runtime.

```
#pragma acc parallel loop private(i,j) tile(8,8)
for(i=0; i<rows; i++) {
   for(j=0; j<cols; j++) {
        out[i*rows + j] = in[j*cols + i];
   }
}
```

Listing 5.2: Example for OpenACC tiling

The Polyhedral model allows the analysis of dependencies within nested loops. It can identify the tiling optimisations. Pluto [BHRS08] and Polly [GZA+11] are tools that uses the Polyhedral model to apply the tiling optimisation.

The packing is typically associated with the tiling. The tile is copied into consecutive memory positions. OpenACC provides the cache directive. So, the compiler uses this directive to explore data access optimisations (data in registers, software-managed cache, or read-only cache) [LB16]. Our proposal uses annotations that are similar to the directives. However, in our case, annotation partitions the collections in order to redefine the limits of the loops. On the other hand, our approach generates a new method that allows us to create more levels of tiling. As long as the tile size limitation, our approach overcomes this limitation by reading a JVM environment variable to adjust the tile size. Listing 5.2 shows the tiling implementation in OpenACC. Two external loops are inserted by OpenACC that process the problem by tiles.

Our approach uses domain decomposition to apply tiling optimisation. Additionally, the approach allows the developer to use packing and to change processing subdomains.

5.3 Parallelism and privatisation

Java supports the shared memory programming model from the beginning. Currently, Java has a set of constructors with different abstraction levels: threads, tasks and parallel streams. Java also provides concurrent mechanisms. However, in scientific applications, it is common to use thread private data to

optimise the parallel execution. Java does not provide explicit support for this technique, contrary to our approach.

Placing synchronisation in the access to the data has a simple implementation being hidden in a library, as is the case with the *ConcurrentHashMap*. However, its use is often ruled out in scientific applications because it limits the parallelism scalability since each access requires blocking operations. On the other hand, the thread private data technique removes access control while processing the data. At the end of processing, the data are aggregated in order to compute the final result. However, its use is not always possible and might be complex to implement. It forces the developer to define new data structures for each thread and a data reduction operation. OpenMP supports thread private data structures, as well as user-defined data reductions for more complex data types. In our approach, it is possible to specify a set of fields as thread private, reducing the code changes required. Access to thread private data is performed transparently by replacing the data structure in the methods that access each field.

OpenMP uses annotations to introduce parallel execution in (sequential) base codes. OpenMP provides a fork-join execution model. The *parallel for* is one of the most commonly used annotations since it runs loop iterations across multiple threads. OpenACC [WSTaM12] uses primitives similar to OpenMP for developing parallel code to run on accelerators (e.g., GPUs). More recently, OpenMP standard also supports offloading to accelerators (e.g., parallel for).

Our approach partitions the domain to support parallel execution. OpenMP and similar approaches partitions loop having no direct meaning the domain. Therefore, our approach inserts parallelism through domain concepts.

5.3.1 Skeletons

Cole [Col89] has defined that the domain program should use skeletons that are higher-order functions or templates. The skeleton defines the parallel execution strategy. So, the skeletons allow hiding the optimisation details and can be fine-tuned to the execution platform.

For the C language, there are multiple approaches that use Skeletons to insert parallelism. SkePU2 [EK10, ELK18] supports traditional skeletons (map, reduce ...). A new skeleton *call* allows the developer to encapsulate the method called on each platform execution inside the skeleton (e.g., CPU or GPU execution). The Threading Building Blocks(TBB) has the flow-graph skeleton, defining task pipeline, where more complex compositions are possible. TBB optimises data access by making parallel allocators available and defining data alignments to avoid false sharing. On the other hand, the TBB allows the developer to associate data with tasks, making the task processing always on the same thread. Finally, TBB has a

set of parallel containers. STAPL allows scaling applications upto 100000 cores [Rau15]. Additionally, the frameworks have a set of containers compatible with the STL [AJR+01]. Musket [RWK19] introduces mechanisms for the efficient composition of skeletons. One example is the skeletons fusion, which translates into a fusion of loops. Our approach does not support the optimisation of this kind. However, this type of optimisation can be specified by the developer. The developer can use the map with a composition of methods.

Habanero-Java [CZSS11] and JaSkel [FSP06, SP07] are skeleton frameworks for Java language. Habanero-Java extends the language by adding data structures and parallelism support. JaSkel uses the OOP class hierarchy to provide different skeleton implementations (e.g., seq, parallel, ...). JaSkel supports the farm and pipeline. The Java 8 streams can also be considered as skeletons to the shared memory platform. These frameworks enable parallel executions but do not support tiling optimisation.

Our approach supports three skeletons: map, reduce and *gSplitMapJoin*. The first two patterns are common skeletons. As in the Cole definition, these are two higher-order functions that the developer uses and where the parallel execution details are hidden. *gSplitMapJoin* introduces a new concept, internally, higher-order functions are used to hide the details. However, the developer creates an annotation that adapts the original code to the skeleton. In the traditional skeleton approach, all code is written as a skeleton parameter, in our approach, the code is written normally and the annotation adapts it to the skeleton.

5.4 Summary

Table 5.2 summarises the optimisations that are supported by each approach. The proposed approach is the approach that provides support for the largest set of optimisations.

In this chapter, we discussed different approaches that enable layout change. For Java, there are projects that use the virtual machine, which, for simple objects, transforms AoP layout into AoS. For the C language, there are approaches that allow the layout to change from AoS to SoA. Our approach uses the AoP layout for development and later provides two layouts (AoP and SoA). The approach allows to support more layouts in the future, but, in this context, only layouts supported natively by Java are provided.

For sorting, we presented two approaches that improve the data locality through the garbage collector. Our option provides a method that sorts the AoP collection, which requires a new call in the domain code. However, this kinds of sorting only works for the AoP layout. However, typically the most efficient layout is the SoA, so this is not one key optimisation to focus on.

The approaches presented for tiling are based on loops. Our approach uses domain decomposition

	Data Layout	Sorting	Packing	Loop Tiling	Loop Fusion	Loop Reorder	Privatisation	Parallelism
JCRNE	✓							
Valhalla	✓							
Wimmer	✓							
SDLT	✓							
ASX	✓							
Sharma	✓							
Hirzel		✓						
Chilimbi		✓						
OpenMP							✓	✓
OpenACC			✓	✓				✓
Mint				✓				✓
Pluto				✓	✓	✓		
Polly				✓	✓	✓		
Java streams								✓
Habanero								✓
TBB								✓
PSTL								✓
GasPar	✓	✓	✓	✓			✓	✓

Table 5.2: Approaches supported optimisations

to decomposes the domain into smaller parts, bringing the optimisation technique closer to the domain. However, this option has a little cost since it requires new structures to support the subdomains. On the other hand, the approach enables the packing optimisation for each subdomain. In short, our approach increases the code abstraction and enables packing optimisation with a small overhead.

Loop Fusion and loop reorder are optimisations based on loops. There are frameworks based on the Polyhedral model that analyse the loops to apply these optimisations. Additionally, this model enables loop tiling.

There are several tools to support parallelism on multiple platforms. In this work, the parallelism explored is thread-based for data parallelism. *Java streams* provide higher-order functions that also enable data parallelism. These functions process a collection with multiples threads. Skeleton-based tools also use higher-order functions to provide parallelism. These approaches usually also allow parallelism in distributed memory. Our approach is also compatible with the use of *Java Streams*.

5.4.1 Decisions

During the thesis, we make decisions regarding approach implementation. Of these decisions, we will highlight two: Aggregation vs Composition and the UML Tool.

Aggregation vs composition

Our approach uses the composition relationship. The collection entities only exist in this context (they only exist if the collection exists). The aggregation relationship with the SoA and AoS layout is complex. One of the problems is what to do when an object is removed from the collection:

- how to know that the object has been deleted, that is, position x does not contain valid values;
- what to do with the existing references.

Both problems have possible resolutions, but their solution implementation is complex, adding overhead to the program. Thus, we chose to support only the composition relationship.

The first problem could be solved, adding an array of *Boolean* that identifies whether the value is valid or not. The second has a more complex resolution since it is potentially necessary to validate all references to the objects in the collection. One way would be to create an object for each entity in the collection, but it would lead to an additional access cost.

UML tool

Using UML to generate the collections library is an approach better integrated into the typical development of complex codes. Thus, the developer reduces the developing cost. Other code generation tools use Java annotations for a similar purpose. However, our option increases the abstraction level in code development. On the other hand, the annotations can simplify the development in simple cases. The example 5.3 shows the MD example with annotations. In this case, the interface can be annotated with the *gCollection*. This annotation generates the same classes and interfaces as the UML tool. *@gPrivate* allows to define thread private structures and has a parameter that identifies the split that will be created to use the thread private data.

The annotation option was not considered in the development, since the complex software uses the domain model in the development process.

```
package ParticleCollection;

import gCollection.*;

@gCollection
public interface Particle extends gCopy{

  public gVector getPosition();
  public gVector getVelocity();
  @gPrivate(name="PrivateForce")
  public gVector getForce();
  public void setPosition( gVector position );
  public void setVelocity( gVector velocity );
  public void setForce( gVector force );
}
```

Listing 5.3: Approach with annotation

Chapter 6

Conclusion and future work

6.1 Conclusion

This thesis introduced an approach that supports the most common data locality optimisations (optimised layout and tiling) while maintaining domain abstractions in the code.

The approach provides a new programming interface that supports multiple data layouts while keeping domain abstractions. The programming interface also enables the usual data locality optimisations in the last development step. Moreover, optimisations can be activated or deactivated easily (pluggable optimisations).

The new programming interface turned possible to create a new methodology to develop scientific applications, enabling a two-step development process: first, developers start by writing abstract domain code without being concerned with performance issues. The main focus of this development step is on program abstraction and correctness; second, after implementing a fully functional application, the developers improve the application performance by introducing optimisations that map the abstract code into a specific platform.

The approach is based on two mechanisms to support this methodology: first, a programming interface provides a data API compatible with Java collections, but it can also encapsulate more efficient data layouts for scientific applications (e.g., SoA); second, a mechanism based on the domain decomposition allowing expressing several kinds of optimisations, namely, tiling, packing and parallelism, implemented through Java annotations.

This second mechanism allowed us to unify several optimisations that are based on domain decomposition. The developer defines how to create the subdomains and selects the optimisations. The tiling is supported by defining how to decompose the domain. The packing is based on tiling and is defined by

an annotation parameter (copy a subdomain into a new collection). Parallelism is also supported through another annotation parameter, that defines how to process subdomains in parallel.

The mechanisms turned possible to support a complete set of locality optimisations. There are other approaches that only support a subset of those optimisations. For instance, SDLT enables efficient data layouts. Others approaches enable other optimisations for a specific step of program development (tiling, parallel execution). The proposed approach allowed to combine this set of optimisations.

The main challenge of this work was how to improve the programmability of scientific applications without losing performance. The support for the development of more abstract code was accomplished by using OOP. However, this type of programming can have a significant overhead in performance, which is incompatible with the high performance required by scientific applications. The developed data API plays a central role in this goal, but some OOP features are not supported since they are not essential in scientific applications. Namely, the approach does not support data polymorphism and object identity (i.e., assignment in collections are performed by copy, which is different from the Java model).

JECoLi is a good example of a scientific framework privileging abstract code over performance. In this framework, the generic collections were replaced by collections complying with the proposed data API. This made it possible to use the SoA layout leading to a 1.6X performance improvement. In this case, distinct collections for each data type were generated by the framework. The changes performed to the JECoLi did not compromise the abstract code and were minimal: first, the creation of collections; second, other changes due to the use of primitive objects.

The developed framework implementation provided access to the data through a proxy. This implementation technique made it possible to use the same domain methods while applying tiling, packing and parallel execution. Also, it was possible to use the thread private data technique by defining only a method to reduce private thread data.

The approach provided an improvement in performance in the various case studies. In the MD case study, a large set of optimisations was assessed, resulting in an improvement of the execution time of 55X. The number of instructions executed in simple cases was similar to traditional techniques with only one or two additional instructions per element processed, which was not relevant in terms of performance. In the MD case study, we created a new code version, using a model of entities closer to the domain without losing performance in the most efficient layout. In the AoP layout, there is an additional cost caused by the cost of the layout structure.

In all case studies, the SoA layout has better performance. Changing to SoA layout is complex: changing access to collections, changing methods API, etc. *GasPar* approach reduced the cost by avoiding a change in API methods. The overheads introduced by the approach were negligible in the most efficient

layouts.

The dynamic compilation of Java allowed optimising the execution of the application. The developer uses the provided interfaces to write the domain code. These interfaces are replaced by concrete objects at runtime, leading to potential calls to virtual methods. However, the Java compiler is able, at runtime, to generate a version of the code where calls to these methods are expanded in-line. Thus, the efficiency of the proposed approach relies on the ability of modern Java compilers to remove the potential overheads of using interfaces, temporary objects (e.g., iterators), etc.

6.2 Future Work

Throughout the work developed, some decisions had to be made, that did not allow to explore all the approach potentialities, leaving some lines of research to be explored. The three main lines of investigation pointed for future work are discussed in this section: supporting polymorphism in the SoA layout; new data layouts (e.g., flat AoP); and improvement of parallelism support (implementation of thread private data mechanisms for other data types, and parallelism support on other platforms).

6.2.1 Polymorphism

The polymorphism support in the SoA layout is complex, however for AoP layout, it is simple. The current tool does not support polymorphic collections. It would be possible to support the polymorphism in the AoP however, to standardise, the tool does not provide it.

A solution to support inheritance with the SoA layout maps classes into tables like in hibernate [BK05]. In hibernate three types of strategy are supported: the first, creates a table per concrete class; in the second, one table has all classes data; and the third, creates one table by class/interface. In our case, the database tables are replaced by *gCollection*s. The strategy that best fits our case is the third one, which maintains the entities order in the parent collection without needing additional fields. In this strategy, calling the superclass methods does not introduce overhead because the values are stored in the collection representing the superclass. However, with respect to the rewritten methods, it is necessary to test the data type of the object and to consult the collection representing its subclass, which introduces an additional cost.

Figure 6.1 shows a possible class diagram with inheritance support in the approach. In the diagram there are *Car* and *Vehicle* entities that will be stored in collections. *Vehicle* is a polymorphic class that, in this case, has a single derived entity that is a *Car*. The common fields are stored in arrays in the *Vehicle*

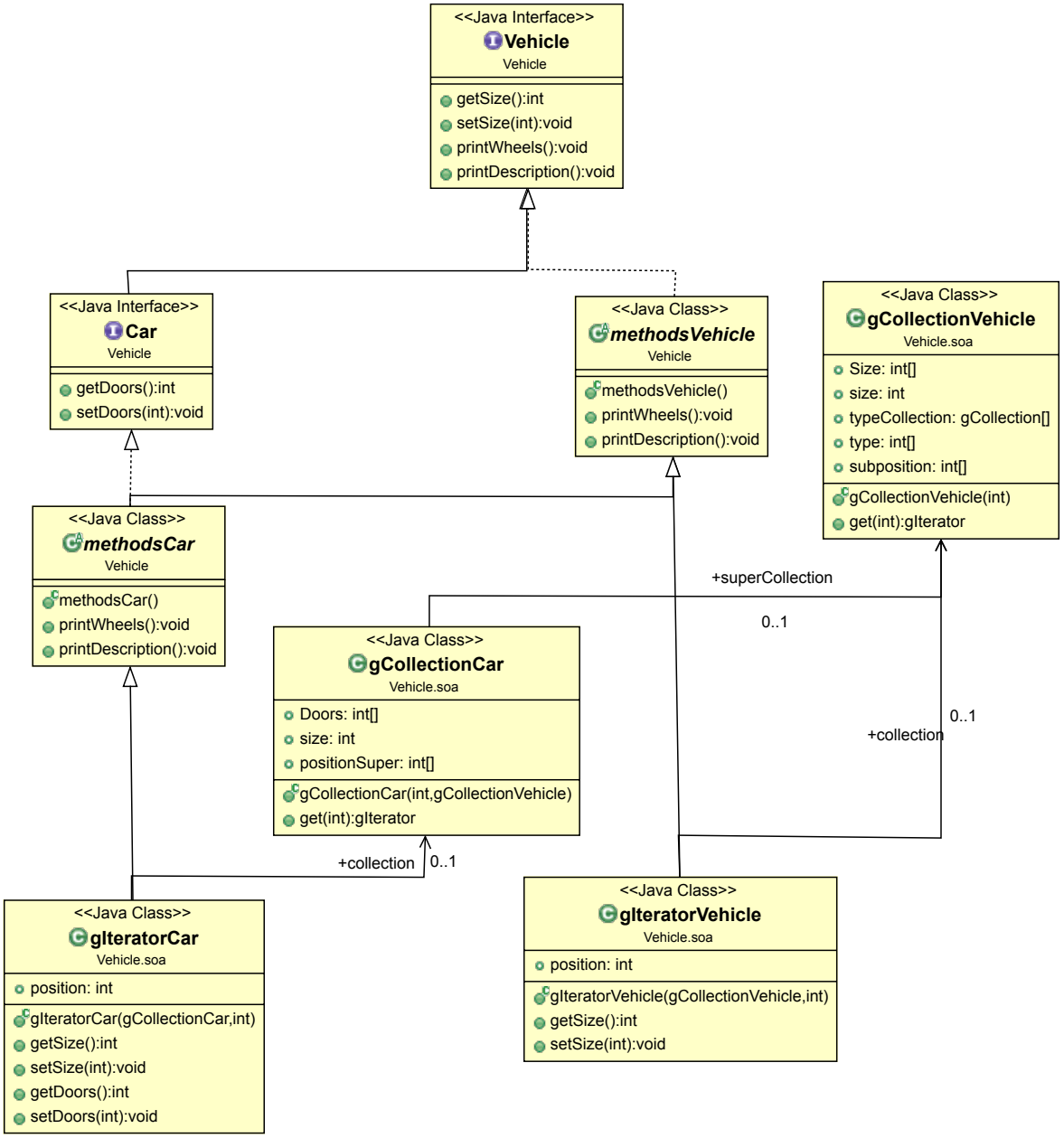


Figure 6.1: Class diagram to support polymorphism

collection. The exclusive properties of the cars are stored in the *gCollectionCar*. In both collections, there is a reference to the position occupied by each entity in the other collection. When the developer needs to process all vehicles, the developer uses an iterator for vehicles, making the *Vehicle* interface methods available (e.g., *printWheels()*). These methods are written with the access methods available in the vehicle

interface (e.g., *getSize()*). If the developer needs to know the vehicle type the developer must use the *getType()* method to find out what vehicle type so that the developer can later cast for the respective type.

6.2.2 Data layouts

Within the scope of this dissertation, the approach provides two distinct layouts. These two layouts supported the validation of the approach. The discussion presented suggests new layouts. One of the proposed layouts improves AoS implementation in order to allow vectorisation. On the other hand, Sharma presents a tool that generates a hybrid layout according to the use of the fields. A future possibility is to extend the tool to include the layout for vectorisation and adapt the tool in order to generate more complex layouts (e.g., hybrid layouts). Java by default does not allow the use of the AoS layout. Removing this limitation implies the usage of Java unsafe.

The AoP layout with composite structures increases the intrinsic AoP overhead, as it implies multiple accesses to several memory zones. In the current implementation, one additional memory access is required for each nested structure. A possible optimisation is to join the structures into a single one, thus avoiding those additional accesses. The programmability would be improved by using the composite structures and polymorphism, but with the same performance.

6.2.3 Parallelism

The framework supports parallel execution by processing collections in shared memory. This section discusses how to extend the tool to support more alternatives: improving thread private data support, support for distributed memory and accelerators.

Thread private data

The approach provides thread private data for collections. However, the current tool does not provide support for a single object or single primitive variable. The approach can combine this mechanism with Java built-in mechanisms. In the MD, the developer controls access to variables using *synchronised*. In that case, the impact on performance is small since the update of the variables is only n times, but algorithm complexity is n^2. For the *force* vector, the developer uses the thread private data mechanism.

gSpitMapJoin defines the parameters that manage the thread private data. The idea is to extend the mechanism so that it is also possible to create private objects for each thread. The use of the mechanism

with primitive data is not relevant since the primitive parameters are passed by value and not by reference. It means that the method will not update the external variable.

In the domain model, all entities relevant to the domain are represented. Typically the data required to be local to each thread is associated with domain entities that are present in the domain model. Therefore, it could be possible to use the UML tool to generalise a method that duplicates and reduces common types of thread-local data.

The developer just has to define in *gSplitMapJoin* (listing 6.1) how each parameter is decomposed and at the end how the data is joined. To divided each parameter, the developer would use the methods that were generated by the tool (in this case the *PrivateData* method will be used). To join each parameter, the developer defines which reduction is defined in his code (as an example of adding two values, in this case, the *reduceData* method defined in class my is used).

```
@gSplitMapJoin(name="f1", map="mapParallel", split={"PrivateData", "none"}, join={"my::
    ReduceData", "none"})
public static void f(My_object p1, gCollection p2) {
   (...)
}
```

Listing 6.1: *gSplitMapJoin*: decomposition the problem in multiple small problems

Expansion of the mechanism allows the developer to use the thread-local mechanism with greater simplicity, hiding irrelevant implementation details.

Distributed memory

The current implementations of distributed memory for Java offer the possibility to use a lot of computational resources. For example, RMI allows the developer to access objects that are on a remote machine. However, these approaches have a significant overhead on performance, and they are not transparent to the developer. The *gSpliMapJoin* mechanism forces the developer to describe all data used. Based on the description, it is possible to optimise the object's access.

On the other hand, there are libraries that are commonly used in HPC with Java support, as is the case with the OpenMPI implementation. MPI is a message passing library that allows developers to implement their code that can be executed in distributed memory system. It is the most used platform for HPC development in the distributed memory. However, the use of MPI has a significant impact on the way of writing the code. The messages that are exchanged between the different processes appear in the middle of the domain code.

The idea is to add support for distributed memory in the *gSplitMapJoin* engine with MPI. The developer will use the mechanism as in the shared memory with some limitations: all input data must be given as parameters, variables with concurrent accesses must use the thread private data mechanism (to specify a reduce operation). The input data has to be parameters in order to be able to identify the data that will be sent to each process. Concurrency control mechanisms should not be used, as they were developed for shared memory. The reduce operation is the same as in the shared memory, with its internal implementation updated for distributed memory.

The developer writes the code, as shown in the example (listing 6.2). The main difference is the map, for this case, the developer uses the *mapParallelDistribute*. For the first parameter, it sends a data copy to all processing units. At the end of the processing, the data are reduced by the *my::ReduceData*. For the second parameter, it needs to send all data to all process units.

```
@gSplitMapJoin(name="f1", map="mapParallelDistribute", split={"PrivateData","none"}, join={"my
    ::ReduceData","none"})
public static void f(gCollection p1, int p2) {
   (...)
}
```

Listing 6.2: *gSplitMapJoin* to distribute memory

GPU

The use of accelerators for processing has gained space nowadays due to its parallel processing capacity. However, its development model is complex, creating difficulties of implementation to the developer. There are currently many tools that attempt to address this problem. Most of the tools are developed for C and C ++.

Aparapi [1], Rootbeer [PSFW12] and Tornado [CFP+18] are approaches that support GPU in Java language. The IBM introduced a JVM implementation that supports stream-like development [IHKS15], where the kernel code is replaced by a lambda method on a stream of integers. All these frameworks do not support memory allocation and objects on GPU code. JaBEE [ZLG12] is a research project that aims to fully support object-oriented development on GPUs. The JaBEE still requires a kernel-based code (like Aparapi) and the project faced several limitations since it depends on the VM internal data layout and was implemented on a specific VM (VMKit).

One more fundamental problem is that objects/structures are not efficient on GPU. In GPUs, it is possible to have multiple accesses to memory in a single transaction (memory coalescing) for this, it

[1] http://aparapi.com/

is necessary to satisfy some requirements. The SoA layout promotes one of these requirements, thus improving access to data. The AoS layout disables the memory coalescing, requiring more transactions to access the same data [KWM16]. *Gaspar* provides SoA layout for data collections that is essential to obtain good performance on GPUs.

In order to improve *Gaspar*, it would be used one of these Java approaches to create the GPU version, such as Aparapi. The current implementation of Aparapi imposes many limitations to the Java code. Aparapi only supports a single object instance on the GPU (i.e., an instance of the Kernel class), including object instance variables and methods. Thus, in order to support GPU execution, the tool would generate a single class (Kernel class extension) to implement all three interfaces (class *gGpuParticle* in figure 6.2 implements the collection, iterator and data entity). The *gCollection map* could be executed to GPUs (see listing 6.3) using Aparapi API.

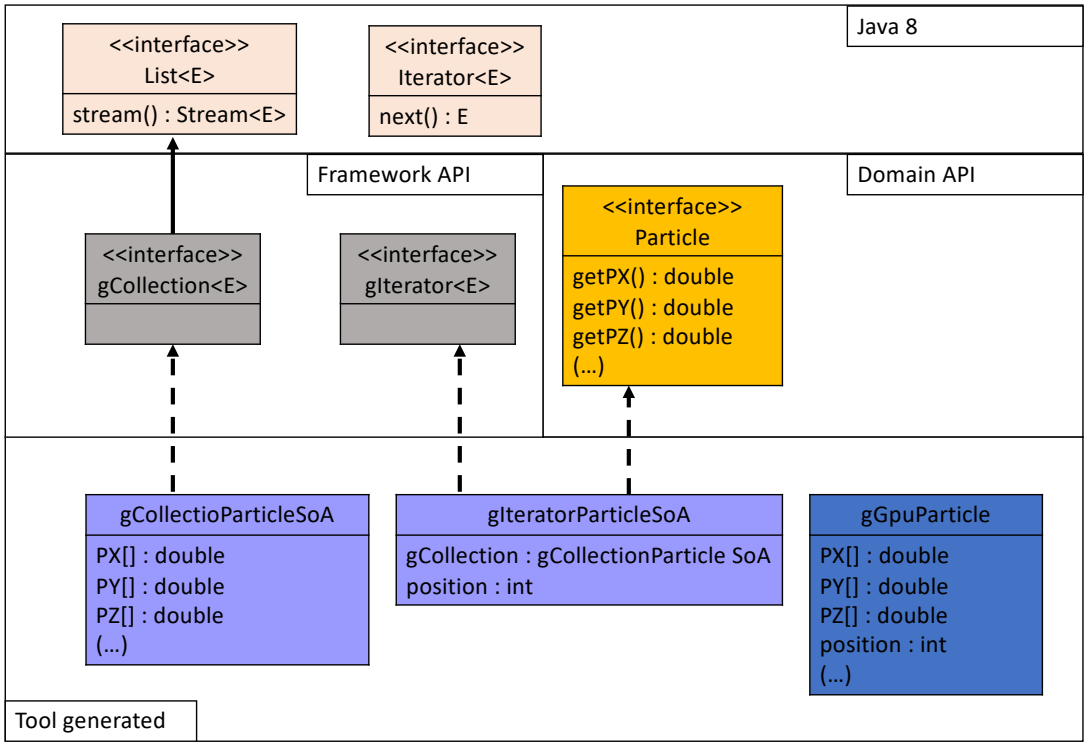


Figure 6.2: Class diagram to support GPU

The *gGpuParticle* class would implement all the required functionality to execute the code on a GPU (see listing 6.4). The class should include the *gCollection* and *gIterator* getter and setter methods and the methods to execute.

In the Aparapi, each GPU thread executes the run method of the kernel class. In this example, each

```
// gpu call
gCollection.foreach(gGpuParticle::move);

// foreach implementation
public void foreach(voidFunction<T> f){
   Kernel aux = new gGpuParticle(this);
   aux.execute(Range.create(this.size()));
   aux.dispose();
}
```

Listing 6.3: Implementation of the foreach method on GPU

```
// gpu call
public class gGpuParticle extends Kernel {

   // gCollection<Particle>
   int size;
   double px[];
   (...)

   // Iterator<Particle>
   double getPX() { return(px[getGlobalId()])};
   (...)

   // stream operation
   void move() {
      double x = getPX();
      (...)
   }
   public void run() {
      move();
   }
}
```

Listing 6.4: Implementation of the move method on GPU

GPU thread will apply the stream method (particle move in this case) to one element of the *gCollection*, so each thread gathers the element corresponding to its global id. This implementation relies on the Aparapi to identify the data to copy to/from GPU memory.

Bibliography

[AJR+01] Ping An, Alin Jula, Silvius Rus, Steven Saunders, Tim Smith, Gabriel Tanase, Nathan Thomas, Nancy Amato, and Lawrence Rauchwerger. Stapl: An adaptive, generic parallel c++ library. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 193–208. Springer, 2001.

[APC+96] Joel Auslander, Matthai Philipose, Craig Chambers, Susan J Eggers, and Brian N Bershad. Fast, effective dynamic compilation. *ACM SIGPLAN Notices*, 31(5):149–159, 1996.

[BHRS08] Uday Bondhugula, Albert Hartono, J Ramanujam, and P Sadayappan. A practical and fully automatic polyhedral program optimization system. In *ACM SIGPLAN PLDI*, volume 10, 2008.

[BK05] Christian Bauer and Gavin King. *Hibernate in action*, volume 4. Manning Greenwich CT, 2005.

[BSW+00] M. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. Benchmarking java grande applications. In *Proceedings of the Second International Conference on The Practical Applications of Java, Manchester, UK*, pages 63–73, 2000.

[BVDGN06] Paolo Bientinesi, Robert Van De Geijn, and FLAME Working Note. *Representing dense linear algebra algorithms: A farewell to indices*. Computer Science Department, University of Texas at Austin, 2006.

[Car02] Carlos Carvalho. The gap between processor and memory speeds. In *Proc. of IEEE International Conference on Control and Automation*, 2002.

[CASL17] Diego Costa, Artur Andrzejak, Janos Seboek, and David Lo. Empirical study of usage and performance of java collections. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, pages 389–400. ACM, 2017.

[CFP+18] James Clarkson, Juan Fumero, Michail Papadimitriou, Foivos S Zakkak, Maria Xekalaki, Christos Kotselidis, and Mikel Luján. Exploiting high-performance heterogeneous hardware for java programs using graal. In *Proceedings of the 15th International Conference on Managed Languages and Runtimes, ManLang*, volume 18, 2018.

[CGS+99] Jong-Deok Choi, Manish Gupta, Mauricio Serrano, Vugranam C Sreedhar, and Sam Midkiff. Escape analysis for java. *Acm Sigplan Notices*, 34(10):1–19, 1999.

[CL99] Trishul M Chilimbi and James R Larus. Using generational garbage collection to implement cache-conscious data placement. *ACM SIGPLAN Notices*, 34(3):37–48, 1999.

[CLPT02] S. Chatterjee, A. R. Lebeck, P. K. Patnala, and M. Thottethodi. Recursive array layouts and fast matrix multiplication. *IEEE Transactions on Parallel and Distributed Systems*, 13(11):1105–1123, 2002.

[Col89] Murray I Cole. *Algorithmic skeletons: structured management of parallel computation*. Pitman London, 1989.

[CZSS11] Vincent Cavé, Jisheng Zhao, Jun Shirako, and Vivek Sarkar. Habanero-java: the new adventures of old x10. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 51–61. ACM, 2011.

[EK10] Johan Enmyren and Christoph W Kessler. Skepu: a multi-backend skeleton programming library for multi-gpu systems. In *Proceedings of the fourth international workshop on High-level parallel programming and applications*, pages 5–14, 2010.

[ELK18] August Ernstsson, Lu Li, and Christoph Kessler. Skepu 2: Flexible and type-safe skeleton programming for heterogeneous parallel systems. *International Journal of Parallel Programming*, 46(1):62–80, 2018.

[EMR09] P. Evangelista, P. Maia, and M. Rocha. Implementing metaheuristic optimization algorithms with jecoli. In *2009 Ninth International Conference on Intelligent Systems Design and Applications*, pages 505–510, Nov 2009.

[FSP06] JF Ferreira, JL Sobral, and AJ Proenca. JaSkel: A Java skeleton-based framework for structured cluster and grid computing. In *Sixth IEEE International Symposium on Cluster Computing and the Grid, 2006. CCGRID 06*, volume 1, 2006.

[FSS13] Nuno Faria, Rui Silva, and Joao L Sobral. Impact of data structure layout on performance. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 116–120. IEEE, 2013.

[GHJV93] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In *European Conference on Object-Oriented Programming*. Springer, 1993.

[Gra02] Mark Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML, Volume 1*. Wiley, New York, 2002.

[GZA+11] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. Polly-polyhedral optimization in llvm. In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, page 1, 2011.

[HCM14] Nassim A Halli, Henri-Pierre Charles, and Jean-François Mehaut. Performance comparison between java and jni for optimal implementation of computational micro-kernels. *arXiv preprint arXiv:1412.6765*, 2014.

[Hir07] Martin Hirzel. Data layouts for object-oriented programs. In *Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, SIGMETRICS '07, pages 265–276, New York, NY, USA, 2007. ACM.

[HKH+16] Samir Hasan, Zachary King, Munawar Hafiz, Mohammed Sayagh, Bram Adams, and Abram Hindle. Energy profiles of java collections classes. In *Proceedings of the 38th International Conference on Software Engineering*, pages 225–236. ACM, 2016.

[Hoa62] Charles AR Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[Hor16] Cay S Horstmann. *Big Java, Binder Ready Version: Early Objects*. John Wiley & Sons, 2016.

[IHKS15] Kazuaki Ishizaki, Akihiro Hayashi, Gita Koblents, and Vivek Sarkar. Compiling and optimizing java 8 programs for gpu execution. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 419–431. IEEE, 2015.

[Int16] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, 2016.

[JJT+07] Hrvoje Jasak, Aleksandar Jemcov, Zeljko Tukovic, et al. Openfoam: A c++ library for complex physics simulations. In *International workshop on coupled methods in numerical dynamics*, volume 1000, pages 1–20. IUC Dubrovnik Croatia, 2007.

[JMF18] Sylvain Jubertie, Ian Masliah, and Joel Falcou. Data layout and simd abstraction layers: decoupling interfaces from implementations. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*, pages 531–538. IEEE, 2018.

[JRS16] James Jeffers, James Reinders, and Avinash Sodani. *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Morgan Kaufmann, 2016.

[JTSE10] Aamer Jaleel, Kevin B. Theobald, Simon C. Steely, Jr., and Joel Emer. High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 60–71, New York, NY, USA, 2010. ACM.

[Kah65] W. Kahan. Pracniques: Further remarks on reducing truncation errors. *Commun. ACM*, 8(1):40–, January 1965.

[KW03] Markus Kowarschik and Christian Weiß. An overview of cache optimization techniques and cache-aware numerical algorithms. In *Algorithms for memory hierarchies*, pages 213–232. Springer, 2003.

[KWM16] David B Kirk and W Hwu Wen-Mei. *Programming massively parallel processors: a hands-on approach*. Morgan kaufmann, 2016.

[LB16] Ahmad Lashgar and Amirali Baniasadi. Openacc cache directive: Opportunities and optimizations. In *2016 Third Workshop on Accelerator Programming Using Directives (WACCPD)*, pages 46–56. IEEE, 2016.

[Len14] Gregory Lento. Optimizing performance with intel advanced vector extensions. *White Paper of Intel-2014*, 2014.

[Lev09] David Levinthal. Performance analysis guide for intel core i7 processor and intel xeon 5500 processors. *Intel Performance Analysis Guide*, 30:18, 2009.

[LRW91] Monica D Lam, Edward E Rothberg, and Michael E Wolf. The cache performance and optimizations of blocked algorithms. *ACM SIGOPS Operating Systems Review*, 25(Special Issue):63–74, 1991.

[LW94] A.R. Lebeck and D.A. Wood. Cache profiling and the spec benchmarks: A case study. *Computer*, 27(10):15–26, 1994.

[MBZ+13] Deepak Majeti, Rajkishore Barik, Jisheng Zhao, Max Grossman, and Vivek Sarkar. Compiler-driven data layout transformation for heterogeneous platforms. In *European Conference on Parallel Processing*, pages 188–197. Springer, 2013.

[MCWK99] John Mellor-Crummey, David Whalley, and Ken Kennedy. Improving memory hierarchy performance for irregular applications. In *Proceedings of the 13th international conference on Supercomputing*, pages 425–433. ACM, 1999.

[Med19] Bruno Silvestre Medeiros. *A framework for heterogeneous many-core machines*. PhD thesis, Universidade do Minho, 2019.

[Met02] Steven John Metsker. *The design patterns java workbook*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[MRR12] Michael McCool, Arch Robison, and James Reinders. *Structured parallel programming: patterns for efficient computation*. Elsevier, 2012.

[MSS15] B. Medeiros, R. Silva, and J. Sobral. Gaspar: A compositional aspect-oriented approach for cluster applications. *Concurrency and Computation: Practice and Experience*, 28:n/a–n/a, 10 2015.

[MW06] Steven John Metsker and William C Wake. *Design patterns in Java*. Addison-Wesley Professional, 2006.

[NCL+10] Jiutao Nie, Buqi Cheng, Shisheng Li, Ligang Wang, and Xiao-Feng Li. Vectorization for java. In *IFIP International Conference on Network and Parallel Computing*, pages 3–17. Springer, 2010.

[NFS11] R. Silva N. Faria and J. L. Sobral. Enhancing locality in java based irregular applications. Simpósium de Informática (inForum'11), Coimbra, September 2011, 2011.

[Oak14] Scott Oaks. *Java Performance: The Definitive Guide: Getting the Most Out of Your Code*. " O'Reilly Media, Inc.", 2014.

[OW15] S Osinski and D Weiss. Hppc: High performance primitive collections for java, 2015.

[PGB+06] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice: JAVA CONCURRENCY PRACT _p1*. Pearson Education, 2006.

[PNDN99] P.R. Panda, H. Nakamura, N.D. Dutt, and A. Nicolau. Augmenting loop tiling with data alignment for improved cache performance. *Computers, IEEE Transactions on*, 48(2):142–149, 1999.

[PSFW12] Philip C Pratt-Szeliga, James W Fawcett, and Roy D Welch. Rootbeer: Seamlessly using gpus from java. In *2012 IEEE 14th International Conference on High Performance Computing and Communication & 2012 IEEE 9th International Conference on Embedded Software and Systems*, pages 375–380. IEEE, 2012.

[Rau15] Lawrence Rauchwerger. The stapl skeleton framework. In *Languages and Compilers for Parallel Computing: 27th International Workshop, LCPC 2014, Hillsboro, OR, USA, September 15-17, 2014, Revised Selected Papers*, volume 8967, page 176. Springer, 2015.

[RWK19] Christoph Rieger, Fabian Wrede, and Herbert Kuchen. Musket: a domain-specific language for high-level parallel programming with algorithmic skeletons. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1534–1543, 2019.

[SB85] Mark Stefik and Daniel G Bobrow. Object-oriented programming: Themes and variations. *AI magazine*, 6(4):40–40, 1985.

[SG98] Peter D Sulatycke and Kanad Ghose. Caching-efficient multithreaded fast multiplication of sparse matrices. In *Proceedings of the First Merged International Parallel Processing Symposium and Symposium on Parallel and Distributed Processing*, pages 117–123. IEEE, 1998.

[SHV+98] Vijayaraghavan Soundararajan, Mark Heinrich, Ben Verghese, Kourosh Gharachorloo, Anoop Gupta, and John Hennessy. Flexible use of memory for replication/migration in cache-coherent dsm multiprocessors. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No. 98CB36235)*, pages 342–355. IEEE, 1998.

[SJG92] Per Stenström, Truman Joe, and Anoop Gupta. Comparative performance evaluation of cache-coherent numa and coma architectures. In *Proceedings of the 19th annual international symposium on Computer architecture*, pages 80–91, 1992.

[SKK+13] Kamal Sharma, Ian Karlin, Jeff Keasler, James R McGraw, and Vivek Sarkar. User-specified and automatic data layout selection for portable performance. *Rice University, Houston, Texas, USA, Tech. Rep. TR13-03*, 2013.

[SL95] Alexander Stepanov and Meng Lee. The standard template library. Technical Report 95-11(R.1), HP Laboratories, 1995.

[SLH12] I-Jui Sung, Geng Daniel Liu, and Wen-Mei W Hwu. Dl: A data layout transformation system for heterogeneous computing. In *2012 Innovative Parallel Computing (InPar)*, pages 1–11. IEEE, 2012.

[Sny86] Alan Snyder. Encapsulation and inheritance in object-oriented programming languages. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 38–45, 1986.

[SOK+04] Toshio Suganuma, Takeshi Ogasawara, Kiyokuni Kawachiya, Mikio Takeuchi, Kazuaki Ishizaki, Akira Koseki, Tatsushi Inagaki, Toshiaki Yasue, Motohiro Kawahito, Tamiya Onodera, et al. Evolution of a java just-in-time compiler for ia-32 platforms. *IBM Journal of Research and Development*, 48(5.6):767–795, 2004.

[SP07] João Luís Sobral and Alberto José Proenca. Enabling jaskel skeletons for clusters and computational grids. In *2007 IEEE International Conference on Cluster Computing*, pages 365–371. IEEE, 2007.

[SRS+12] John A Stratton, Christopher Rodrigues, I-Jui Sung, Li-Wen Chang, Nasser Anssari, Geng Liu, W Hwu Wen-mei, and Nady Obeid. Algorithm and data optimization techniques for scaling to massively threaded systems. *Computer*, 45(8):26–32, 2012.

[SS16] Rui Silva and João L Sobral. Gaspar data-centric framework. In *International Conference on Vector and Parallel Processing*, pages 234–247. Springer, 2016.

[Str11] Robert Strzodka. Abstraction for aos and soa layout in c++. In *GPU computing gems Jade edition*, pages 429–441. Elsevier, 2011.

[Sub11] Venkat Subramaniam. *Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors*. Pragmatic Bookshelf, 2011.

[TBK06] Jeyarajan Thiyagalingam, Olav Beckmann, and Paul HJ Kelly. Is morton layout competitive for large two-dimensional arrays yet? *Concurrency and Computation: Practice and Experience*, 18(11):1509–1539, 2006.

[TJYD10] Dan Terpstra, Heike Jagode, Haihang You, and Jack Dongarra. Collecting performance data with papi-c. In *Tools for High Performance Computing 2009*, pages 157–173. Springer, 2010.

[Tro] GNU Trove. High performance collections for java.

[UCB11] Didem Unat, Xing Cai, and Scott B Baden. Mint: realizing cuda performance in 3d stencil methods with annotated c. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.

[vEBKZ76] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10(1):99–127, 1976.

[Vig16] Sebastiano Vigna. fastutil: Fast and compact type-specific collections for java, 2016.

[Weg87] Peter Wegner. Dimensions of object-based language design. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, OOPSLA '87, pages 168–182, New York, NY, USA, 1987. ACM.

[Wei02] Mark Allen Weiss. *Data Structures and Problem Solving Using Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[Wen19] Florian Wende. C++ data layout abstractions through proxy types. In *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 758–767. IEEE, 2019.

[WM08] Christian Wimmer and Hanspeter Mössenböck. Automatic array inlining in java virtual machines. In *Proceedings of the 6th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 14–23, New York, NY, USA, 2008. ACM.

[WSTaM12] Sandra Wienke, Paul Springer, Christian Terboven, and Dieter an Mey. Openacc—first experiences with real-world applications. In *European Conference on Parallel Processing*, pages 859–870. Springer, 2012.

[YRP+07] Kamen Yotov, Tom Roeder, Keshav Pingali, John Gunnels, and Fred Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, pages 93–104, New York, NY, USA, 2007. ACM.

[YSP+98] Kathy Yelick, Luigi Semenzato, Geoff Pike, Carleton Miyamoto, Ben Liblit, Arvind Krishnamurthy, Paul Hilfinger, Susan Graham, David Gay, Phil Colella, et al. Titanium: a high-performance java dialect. *Concurrency and Computation: Practice and Experience*, 10(11-13):825–836, 1998.

[ZLG12] Wojciech Zaremba, Yuan Lin, and Vinod Grover. Jabee: framework for object-oriented java bytecode compilation and execution on graphics processor units. In *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, pages 74–83. ACM, 2012.

Chapter 7

Appendix

7.1 Sum benchmark

```
//Assembler for aop
(...)
mov     0x18(%rdi,%rbp,4),%edx
(...)
mov     %rdx,%r8
shl     $0x5,%r8
test    %r8,%r8
je      0x00002b3f5c3fabd2
mov     0x8(%r8),%r9d
cmp     $0xed08,%r9d
jne     0x00002b3f5c3fac5b
(...)
vaddsd 0x10(%r8),%xmm0,%xmm0
vmovsd %xmm0,0x90(%rcx)
(...)
```

```
//Assembler for gaop
(...)
mov     0x14(%r9,%rcx,4),%r11d
mov     %r11,%r10
shl     $0x5,%r10
vaddsd 0x10(%r10),%xmm0,%xmm0
mov     $0x1da8d3fe00,%r10
vmovsd %xmm0,0x90(%r10)
(...)
```

Listing 7.1: Instructions need to calculate one element in versions: *aop* and *gaop*

```
//Assembler for soa
(...)
vaddsd 0x10(%r11,%rbx,8),%xmm0,%xmm0
(...)
vmovsd %xmm0,0x90(%r10)
(...)
```

```
//Assembler for gsoa load and sum element
(...)
vaddsd 0x10(%r10,%r11,8),%xmm0,%xmm0
(...)
mov    $0x1f2abbdea0,%r8
vmovsd %xmm1,0x90(%r8)
vmovsd %xmm0,0x90(%r8)
(...)
```

Listing 7.2: Instructions need to calculate one element in versions: *soa* and *gsoa*

```
//Assembler for faop
(...)
mov    0x18(%rbx,%rbp,4),%esi
(...)
mov    %rsi,%r10
shl    $0x5,%r10
mov    %r8d,%ecx
add    $0x3,%ecx
mov    %ecx,0xc(%r9)
mov    0x8(%r10),%eax
cmp    $0xed08,%eax
jne    0x00002b7a60824652
mov    %ecx,0x10(%r9)
vaddsd 0x10(%r10),%xmm0,%xmm0
(...)
mov    $0x1d992fa280,%r10
vmovsd %xmm0,0x90(%r10)
(...)
```

```
//Assembler for fgaop
(...)
mov    %ebx,%r10d
add    $0x2,%r10d
mov    %r10d,0xc(%rax)
(...)
mov    0x14(%rcx,%r8,4),%r10d
shl    $0x5,%r10
vaddsd 0x10(%r10),%xmm0,%xmm0
vmovsd %xmm0,0x90(%rbp)
(...)
```

Listing 7.3: Instructions need to calculate one element in versions: *faop* and *fgaop* (instructions)

```
//Assembler for ggaop
(...)
mov    %ecx,%r10d
inc    %r10d
mov    %r10d,0xc(%rax)
movslq %ecx,%r9
mov    0x14(%rbx,%r9,4),%r10d
shl    $0x5,%r10
vaddsd 0x10(%r10),%xmm0,%xmm0
mov    $0x1da8a6d0a0,%r10
vmovsd %xmm0,0x90(%r10)
(...)
```

Listing 7.4: Assembler code for *ggaop*

```
//Assembler for csaop
(...)
mov     0x14(%rcx,%rsi,4),%r10d
mov     %r10,%rdi
shl     $0x5,%rdi
test    %rdi,%rdi
je      0x00002b0be840069d
mov     0x8(%rdi),%r10d
cmp     $0xed08,%r10d
jne     0x00002b0be8400721
(...)
vmovsd 0x10(%rdi),%xmm2
vsubsd %xmm0,%xmm2,%xmm0
vaddsd %xmm2,%xmm1,%xmm1
vaddsd %xmm3,%xmm0,%xmm2
vsubsd %xmm3,%xmm2,%xmm3
vsubsd %xmm0,%xmm3,%xmm0
vmovsd %xmm0,0x18(%rdx)
vmovsd %xmm2,0x10(%rdx)
vmovsd %xmm1,0x20(%rdx)
(...)
```

```
//Assembler for cssoa
(...)
vmovsd 0x10(%r11,%rcx,8),%xmm0
vsubsd 0x18(%r9),%xmm0,%xmm1
vmovsd 0x10(%r9),%xmm2
vaddsd %xmm2,%xmm1,%xmm3
vsubsd %xmm2,%xmm3,%xmm2
vsubsd %xmm1,%xmm2,%xmm1
vmovsd %xmm1,0x18(%r9)
vmovsd %xmm3,0x10(%r9)
vmovsd 0x20(%r9),%xmm2
vaddsd %xmm0,%xmm2,%xmm0
vmovsd %xmm0,0x20(%r9)
movslq %ecx,%rbx
(...)
```

Listing 7.5: Assembler code of *csaop* and *cssoa*

	load object	test null	checkcast	store index	store result	unroll	#I/element
aop	$$	$$	$$	-	$$	8	10.25
gaop	$$	-	-	-	$$	8	6.5
faop	$$	-	$$	$$$	$$	4	13.5
fgaop	$$	-	-	$$	$$	8	7.25
ggaop	$$	-	-	$$	$$	8	9
csaop	$$	$$	$$	-	$$$	4	18.29
csgaop	$$	-	-	$	$$$	1	15.53
saop	$$	-	$$	$$	$$$	1	48.02
sgaop	$$	-	$$	$$	$$$	1	20.52
soa	-	-	-	-	$	16	1.31
gsoa	-	-	-	$	$	16	1.44
fsoa	-	-	-	-	$	16	1.31
fgsoa	-	-	-	$	$	16	1.44
ggsoa	-	-	-	$	$	16	1.44
JCRNE(aop)	$$	-	-	$$	$$	8	6.5
JCRNE(soa)	-	-	-	-	-	16	1.44

Table 7.1: Group of instructions generated

```
//Assembler for saop
(...)
mov     0x10(%rdi,%r10,4),%ebx
mov     %r10d,0x8(%rsp)
mov     %rdi,(%rsp)
mov     0x30(%rsp),%r10
mov     0xc(%r10),%edx
mov     0x14(%r10),%r11d
mov     %r11,%rdi
shl     $0x5,%rdi
mov     0x8(%rdi),%r11d
cmp     $0x6bc28,%r11d
jne     0x00002b4454401496
mov     %rdx,%rax
shl     $0x5,%rax
mov     0x8(%rax),%r11d
cmp     $0xed08,%r11d
jne     0x00002b44544014eb
mov     %rbx,%rdi
shl     $0x5,%rdi
mov     0x8(%rdi),%r10d
cmp     $0xed08,%r10d
jne     0x00002b4454401550
vmovsd  0x10(%rax),%xmm0
vaddsd  0x10(%rdi),%xmm0,%xmm
(...)
```

```
//Assembler for ssoa
(...)
vaddsd 0x10(%r11,%rbx,8),%xmm0,%xmm0
(...)
vmovsd %xmm0,0x90(%r10)
(...)
```

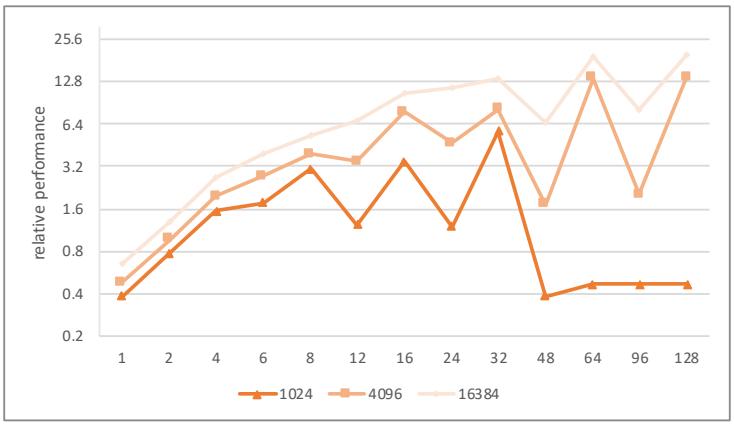
Listing 7.6: Assembler code of *saop* and *ssoa*

7.2 daxpy benchmark

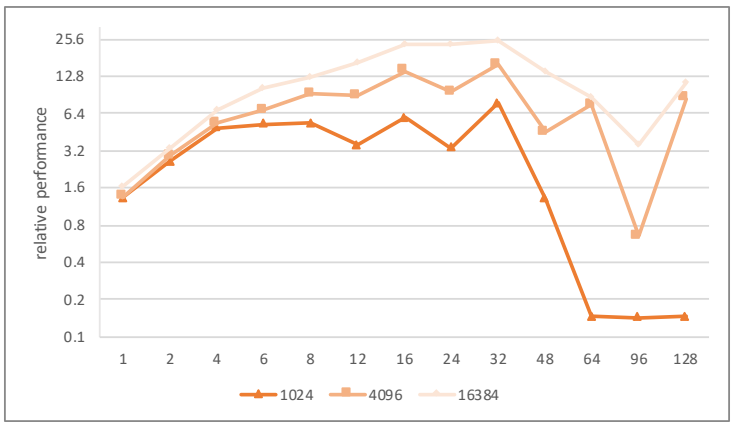
	test null	checkcast	load object	store position	vectorial instructions	unroll	#I/element
aop	rangecheck	$$	2x($$$)	-	-	1	82.00
gaop	$$	-	2x($$)	-	-	4	12.25
faop	rangecheck	$$	2x($$)	2x($$$)	-	1	120.00
fgaop	-	-	2x($$)	$$$	-	1	27.00
ggaop	-	-	2x($$$)	2x($$)	-	1	32.00
aopD	2x($$)	2x($$)	2x($$)	-	-	4	20
faopD	-	2x($$)	2x($$)	2x($$$)	-	1	44
aopJ	$$	$$	$$	-	-	4	12.5
gaopJ	-	-	$$	-	-	8	8.25
faopJ	-	$$	$$	$$$	-	4	14.50
fgaopJ	-	-	$$	$$	-	8	10.125
ggaopJ	-	-	$$	$$	-	8	10
soa	-	-	-	-	+	4x4	1.25
gsoa	-	-	-	-	+	4x4	1.25
fgsoa	-	-	-	2x($$$)	-	1	22.00
ggsoa	rangecheck	-	2x($$)	2x($$$)	-	1	28.00
gsoaJ	-	-	-	-	+	4x4	1.25
fgsoaJ	-	-	-	$	+	4x4	1.31
ggsoaJ	-	-	-	$	+	4x4	1.31
eaop	-	-	2x$$	-	-	1	42
esoa	-	-	-	-	+	4x4	1.25
eaopJ	-	-	$$	-	-	1	36
esoaJ	-	-	-	-	+	4x4	1.25

Table 7.2: Group of instructions generated

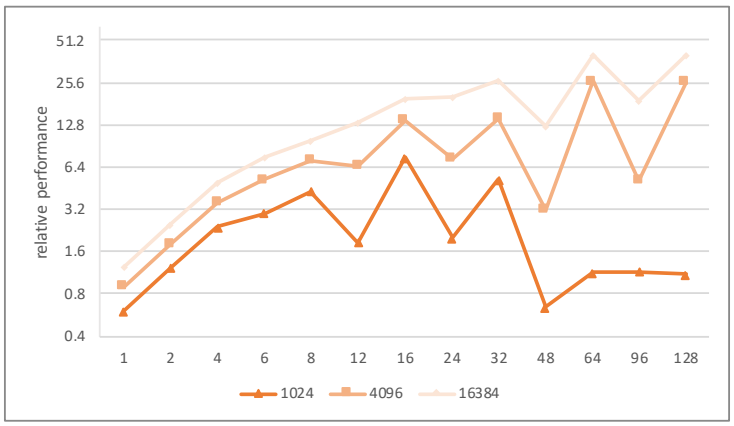
7.3 MM benchmark



(a) tile



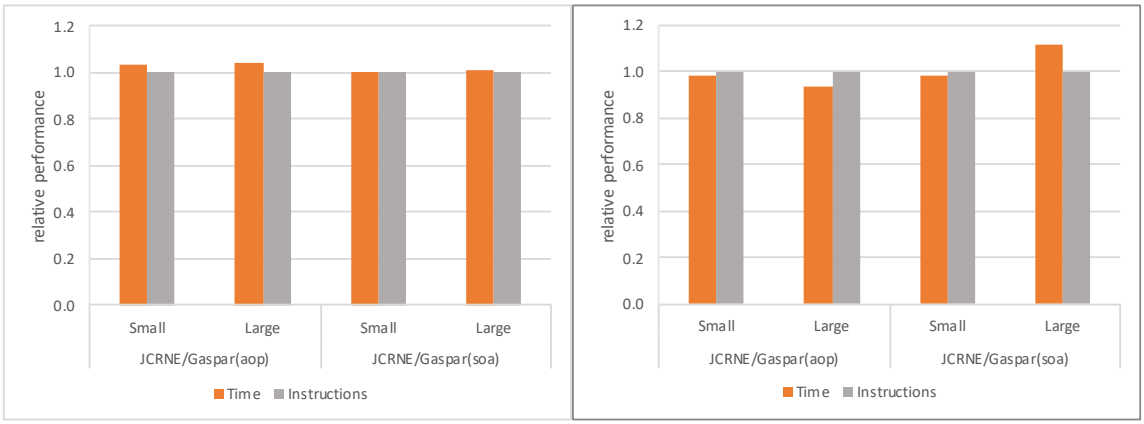
(b) packing



(c) packingOptimise

Figure 7.1: MM - Parallel versions

7.4 Discussion



(a) Sum (b) daxpy

Figure 7.2: Relative performance between JCRNE and *GasPar*