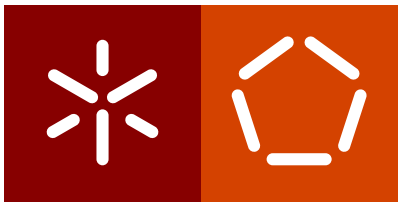
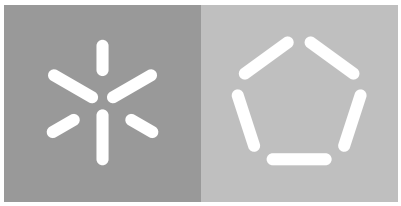


Universidade do Minho
Escola de Engenharia
Departamento de Informática

John Maia

Scheduling Scientific Workloads on an Heterogeneous Server

October 2016



Universidade do Minho
Escola de Engenharia
Departamento de Informática

John Maia

Scheduling Scientific Workloads on an Heterogeneous Server

Master Thesis Dissertation
Master Degree in Informatics Engineering

Supervisor: **Alberto Proença**
External Advisor: **André Pereira**

October 2016

ACKNOWLEDGEMENTS

I begin by thanking my supervisor, Professor Alberto Proença, for the knowledge that he gave me throughout my academic education and for supervising the research and the writing of this dissertation with his high level of thoroughness, that without it, this thesis would not have achieved all its objectives.

To my external advisor, André Pereira, for his enormous patience and ability to make me always go far beyond the goals that I proposed throughout the research. Without the help and contribution, with his extensive expertise in parallel computing paradigms, I wouldn't certainly have achieved the results presented in this thesis.

To the computing facilities provided by the Project "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).

To all my friends, for having encouraged and helped me throughout many moments of frustration that emerged along my research. To Manuela Cunha, for helping me in the multiple moments where I stalled in my research, by providing an interpretation of the issue through a entirely different perspective from mine, which often led me to quickly find a solution. To Bruno Medeiros, for the countless moments of brainstorming, sometimes related to a problem that one of us was facing in its investigation, but sometimes on subjects that were completely outside of our area and context that helped a lot to break the monotony.

To my sister, Marta Maia, who despite being physically so far away, was always interested and supported me along my academic years. I also thank her for always sharing with me her opinion, based on her extensive experience in the academic world, which helped me make important decisions.

Last but definitely not least, I thank my parents, Maria Maia and Camilo Maia, for always supporting me, for believing in me and for giving me the opportunity to have an academic degree, which will certainly help me achieve the future that they have always envisioned for me and to which they both fought.

ABSTRACT

The goal of this dissertation is to explore techniques to improve the efficiency and performance level of scientific applications on computing platforms that are equipped with multiple multi-core devices and at least one many-core device, such as Intel MIC and/or NVidia GPU devices. These platforms are known as heterogeneous servers, which are becoming increasingly popular both in research environments as in our daily gadgets.

To fully exploit the performance capabilities of the heterogeneous servers, it is crucial to have an efficient workload distribution among the available devices; however the heterogeneity of the server and the workload irregularity dramatically increases the challenge.

Most state of the art schedulers efficiently balance regular workloads among heterogeneous devices, although some lack adequate mechanisms for irregular workloads. Scheduling these type of workloads is particularly complex due to their unpredictability, namely on their execution time. To overcome this issue, this dissertation presents an efficient dynamic adaptive scheduler that efficiently balances irregular workloads among multiple devices in a heterogeneous environment.

To validate the scheduling mechanism, the case study used in this thesis is an irregular scientific application that has a set of independent embarrassingly parallel tasks applied to a very large number of input datasets, whose tasks durations have an unpredictable range larger than 1:100. By dynamically adapting the size of the workloads that were distributed among the multiple devices in run-time, the scheduler featured in this dissertation had an occupancy rate of every computing resources over 97% of the application's run-time while generating an overhead well below 0.001%.

RESUMO

O objetivo desta dissertação é o de explorar técnicas que possam melhorar a eficiência e o nível de performance de aplicações cientificas em plataformas de computação que estão equipadas com vários dispositivos multi-core e pelo menos um dispositivo many-core, como por exemplo um Intel MIC e/ou um GPU da NVidia. Estas plataformas são conhecidas como servidores heterogéneos e estão a se tornar cada vez mais populares, tanto em ambientes de investigação como em nossos gadgets diários.

Para explorar completamente as capacidades de desempenho dos servidores heterogéneos, é crucial ter uma distribuição eficiente da carga de trabalho entre os vários dispositivos disponíveis; no entanto a heterogeneidade do servidor e a irregularidade das cargas de trabalho aumentam drasticamente o desafio.

A maioria dos escalonadores mais avançados são capazes de equilibrar eficientemente cargas de trabalho regulares entre dispositivos heterogêneos, embora alguns deles não disponham de mecanismos adequados para cargas de trabalho irregulares. O escalonamento desse tipo de cargas de trabalho é particularmente complexo devido à sua imprevisibilidade, nomeadamente ao seu tempo de execução. Para superar este problema, esta dissertação apresenta um escalonador dinâmico e adaptativo que equilibra de forma eficiente cargas de trabalho irregulares entre vários dispositivos de uma plataforma heterogénea.

Para validar o escalonador, o caso de estudo utilizado nesta tese é uma aplicação científica irregular que possui um conjunto de tarefas independentes, que são embaraçosamente paralelas, aplicadas a um grande número de conjuntos de dados, cujas tarefas têm durações com um nível de imprevisibilidade maior do que 1:100. Ao adaptar dinamicamente o tamanho das cargas de trabalho, que são distribuídas entre os vários dispositivos, em tempo de execução, o escalonador apresentado nesta dissertação apresenta uma taxa de ocupação de cada dispositivo acima de 97 % do tempo total de execução da aplicação e tem um peso que é bem abaixo dos 0,001 %.

CONTENTS

1

INTRODUCTION

Most scientific and engineering software requires large amounts of computational resources due to their high computational complexity. Research groups resort to computing clusters to cope with the increasing demand of both computing power and memory bandwidth to run scientific code in a reasonable time frame.

Most scientific code does not efficiently use the available resources in a cluster computing node, as these applications are usually developed by non-computer scientists, which often are self-taught programmers, without the expertise to efficiently exploit the performance of a single CPU core (by exploring ILP[1], SIMD[2] instructions or the multi-level cache hierarchy) or deal with the complexity of current multi-core systems. These issues are more obvious when dealing with heterogeneous servers, where programming for hardware accelerators places more challenges to fully exploit the potential performance from these devices. To maximize the system computing throughput, the software needs to be properly parallelized and the workloads have to be efficiently balanced across the multiple computing units of each device.

Developing efficient multithreaded applications that simultaneously use multi-core and many-core resources is a complex task. The programmer needs to address not only algorithmic issues, but also has to efficiently structure data in memory, and deal with data races while implementing both shared and distributed memory parallelization strategies simultaneously. These are crucial concepts, not only for the efficient execution of the code but also to ensure the correctness of the applications.

To aid the scientific community in dealing with the implicit challenges of developing applications that efficiently explore the performance of heterogeneous servers, this work addresses the implementation of a higher layer of abstraction to deal with the inherent architecture complexities of these platforms.

1 Instruction-Level Parallelism.
2 Single Instruction stream, Multiple Data stream.

1.1 CONTEXT AND GOALS

This thesis focuses in one of the most complex issues when coding for heterogeneous servers: to schedule and balance irregular workloads among multiple devices (multi-core and many-core devices).

To validate the efficiency of the scheduler, a real scientific application from the High Energy Physics (HEP) field was selected, which analyses data collected from sensors during collisions of proton beams in the ATLAS[3] Experiment at the LHC[4] (CERN[5]). The performance evaluation was conducted in a cluster node with two 12 core Xeon devices and two many-core Intel Xeon Phi devices, in four distinct heterogeneous configurations, to assess the adaptability of the scheduler: (i) 1 multi-core + 1 many-core; (ii) 2 multi-cores + 1 many-core; (iii) 1 multi-core + 2 many-cores; and (iv) 2 multi-cores + 2 many-cores.

1.2 MOTIVATION AND CONTRIBUTION

The innate curiosity of human beings has always led us to question the world around us. Our ability to reason brought answers to an innumerable number of questions throughout many centuries of progress that shaped the world as we know it today.

As new and more complex questions emerge demanding a higher reasoning power, researchers resort to complex software to often deal with large amounts of data. Eventually, the questions became more complex and computationally demanding, requiring the need of using computing platforms that were equipped with one or multiple many-core devices capable of accelerating the resolution. These computing platforms are known as heterogeneous platforms and, to efficiently exploit these platforms, a high level of expertise in High Performance Computing (HPC) is required. However, scientific applications are developed by computational scientists, which often are self-taught programmers. This thesis aims to include HPC features into current scientific code to efficiently use heterogeneous servers.

This thesis proposes a efficient lightweight scheduler that aims to balance irregular workloads among heterogeneous devices and that dynamically adapts the amount of work dispatched to each computing device according to their current performance, while maintaining a low overhead.

3 A Toroidal LHC ApparatuS.
4 Large Hadron Collider.
5 Conseil Européen pour la Recherche Nucléaire.

1.3 DISSERTATION STRUCTURE

This dissertation is structured into seven chapters, whose summary is presented below:

I *Introduction:* introduces the topic of this dissertation by briefly presenting the inherent challenges of developing applications that efficiently explore the performance of heterogeneous servers, and introduces the main objective, which is to efficiently schedule irregular workloads among heterogeneous devices. Section 1.2 presents the scientific motivation and the contribution.

II *Heterogeneous Servers:* this chapter presents the concept of an heterogeneous server, introduces the NUMA[6] architecture in the cluster node and briefly introduces how the accelerators operate in distributed memory. This chapter also presents the current multi-core and many-core devices and describes their features.

III *Scheduling Data and Workloads:* this chapter describes the state of the art on the inherent complexity of scheduling data and workloads on heterogeneous platforms. This chapter also presents a list of key frameworks to aid the scheduling in the homogeneous and heterogeneous platforms.

IV *Scientific Workloads:* presents and describes the type of scientific workloads that this work addresses and presents the selected scientific case study for this dissertation.

V *An Adaptive Scheduler for an Heterogeneous Server:* introduces and describes the implemented dynamic adaptive heterogeneous scheduler and its key features to support irregular workloads.

VI *Experimental results and discussion:* presents and describes the selected testbed to validate the efficiency of the scheduler, and discusses the experimental performance results.

VII *Conclusions:* concludes the dissertation with suggestions for future work.

6 Non-Uniform Memory Access.

2

HETEROGENEOUS SERVERS

Current computer servers are becoming highly parallel due the mix of different computing devices, such as multi-core devices and/or many-core devices (NVidia GPU[1] and Intel MIC[2]). A computing device is considered as a set of computing units that may share the same memory address space. The terminology of heterogeneous devices relates to the different characteristics of computing devices in the same computer server, in a distributed memory address space.

Heterogeneous platforms are a new type of computing platform and it is becoming increasingly popular, largely because these systems gain performance not just by adding cores, but also by incorporating specialized units with processing capabilities to handle particular compute intensive tasks.

Taking advantage of the potential performance of current computer servers requires dealing with several issues. The software needs to be properly parallelized to explore the performance of each individual multi-core and many-core device to maximize its computing throughput, the data needs to be efficiently structure in memory and the workloads need to be efficiently balanced among the available devices.

Most of the state of the art schedulers are capable of balancing regular workloads among multi-core and many-core devices, although some lack adequate mechanisms for irregular workloads, since these type of workloads is particularly complex due to their unpredictability and the differences on their execution time. A poor workload distribution can potentially result in an undesired increase of the application run-time.

To ensure that all devices are used as much as possible, it is imperative to implement a scheduling strategy that follows a heuristic that efficiently balances workloads among the multiple devices, reducing the percentage of the application run-time where a device remain in an idle state, i.e. waiting for more work.

1 Graphics Processing Unit.
2 Many Integrated Core architecture.

2.1 NUMA ARCHITECTURE IN SHARED MEMORY

Multi-core devices memory share the same address space, allowing the multiple processing elements to share their data. A single socket system has the memory bank connected to the CPU through a high bandwidth interface. A system with multiple sockets has a memory bank connected to each CPU device, where all devices share the same address space and this memory architecture is known as NUMA. In this memory architecture, processors do not have to be aware where data resides, except that there may be performance penalties related to memory access time.

The memory access time depends on the memory location relative to the processor, i. e. the time that it takes for one of the cores from one of the multi-core devices to read or write data in the memory connected to another multi-core device is substantially greater than the time that it takes to perform the same operation in its device local memory.

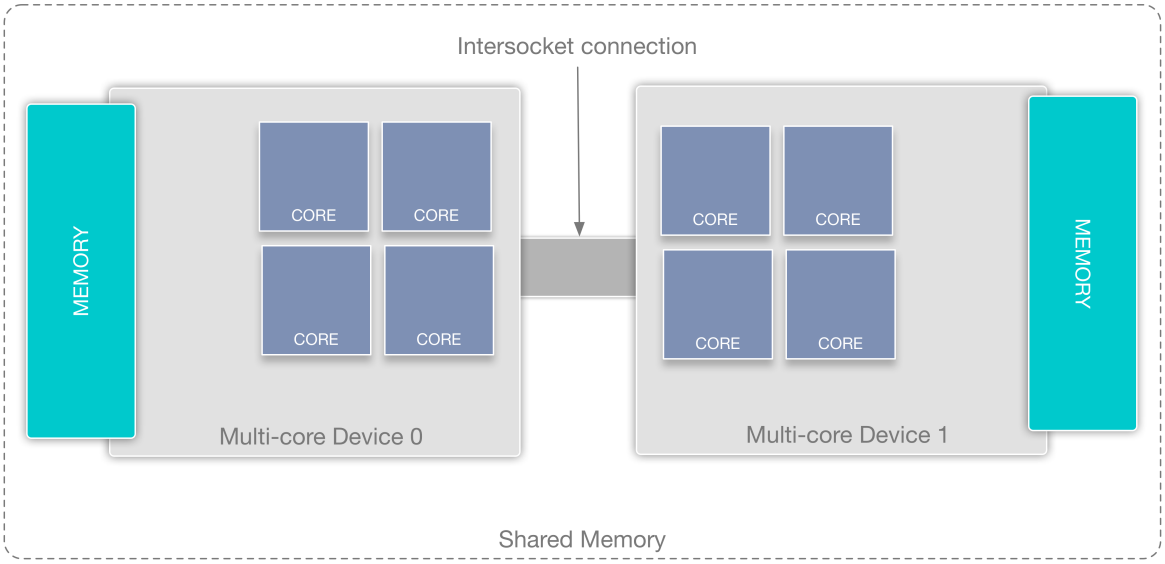


Figure 1: Representation of a NUMA shared memory system

Data locality is crucial when trying to get the most out of the performance capability of a multi-core device. Data intensive applications should maximize the amount of memory access to the closer memory bank. Some of the tools that aid the development of multithreaded applications already have implemented policies, such as the NUMA first-touch placement from OpenMP[20] (Figure 2), which is able to predict and place the data close to physical core from where the thread, responsible of computing the data, will execute.

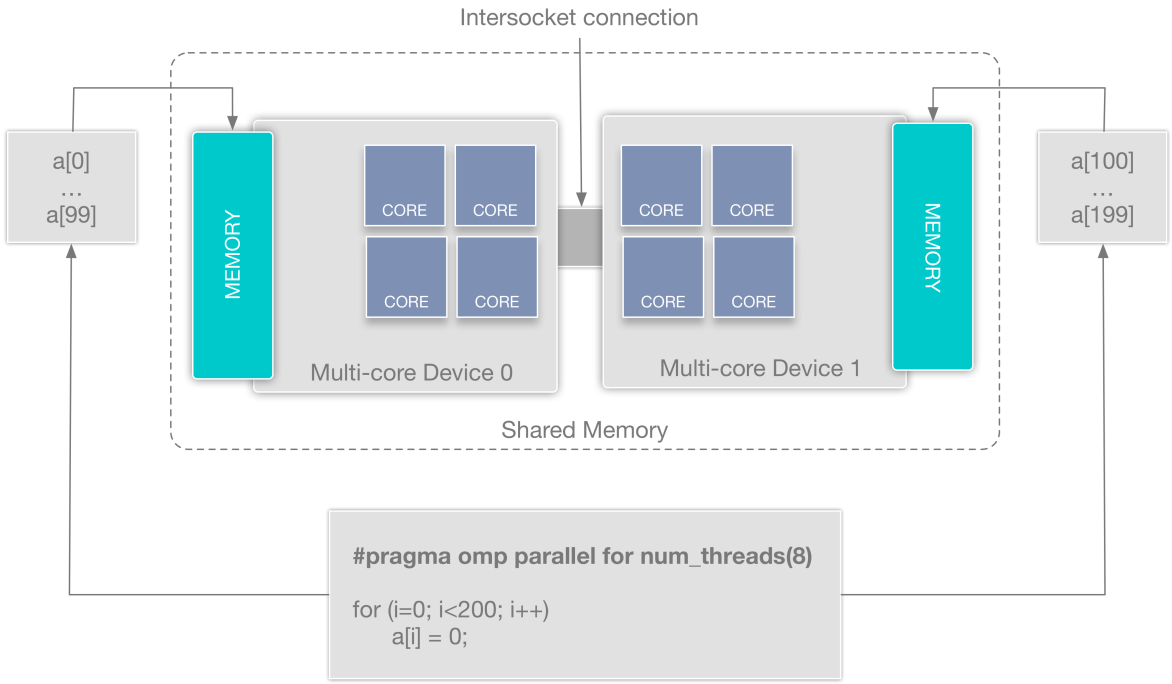


Figure 2: Representation of NUMA first-touch placement from OpenMP

In figure 2 shows that with the first-touch data placement policy in that example, where we have two multi-core devices, OpenMP splits the array `a[]` (workload) evenly among the two devices, the threads that are executing within the cores of CPU0 will firstly access the array elements, that are closer to physical core, which are those that are allocated between the position 0 and 99 and threads from CPU1 will access the elements that are between 100 and 199. Non-uniform memory access, which in this figure represents the CPU0 consuming elements that were assigned to CPU1, or vice versa, will only occur when one of the multi-core devices depletes the elements from the region of the array that was allocated to it.

If one of the multi-core devices consumes all of the available data in its local memory and starts to consume the elements that are allocated in a remote memory region, it is most likely that by that time the amount of data still waiting to be processed, in the other multi-core device memory, will be fairly small and therefore the number of NUMA accesses should also be reduced.

2.2 ACCELERATORS IN DISTRIBUTED MEMORY

Hardware accelerators (also known as many-core devices) have their own local memory. Computational tasks can only operate on local data, and if remote data is required, the software developer has to explicitly handle data transfers to the many-core device memory through a high latency PCIe connection. This requires a deeper knowledge of the program data requirements, since the software developer, to implement algorithms that minimise remote data accesses. This memory architecture is know as distributed memory (Figure 3).

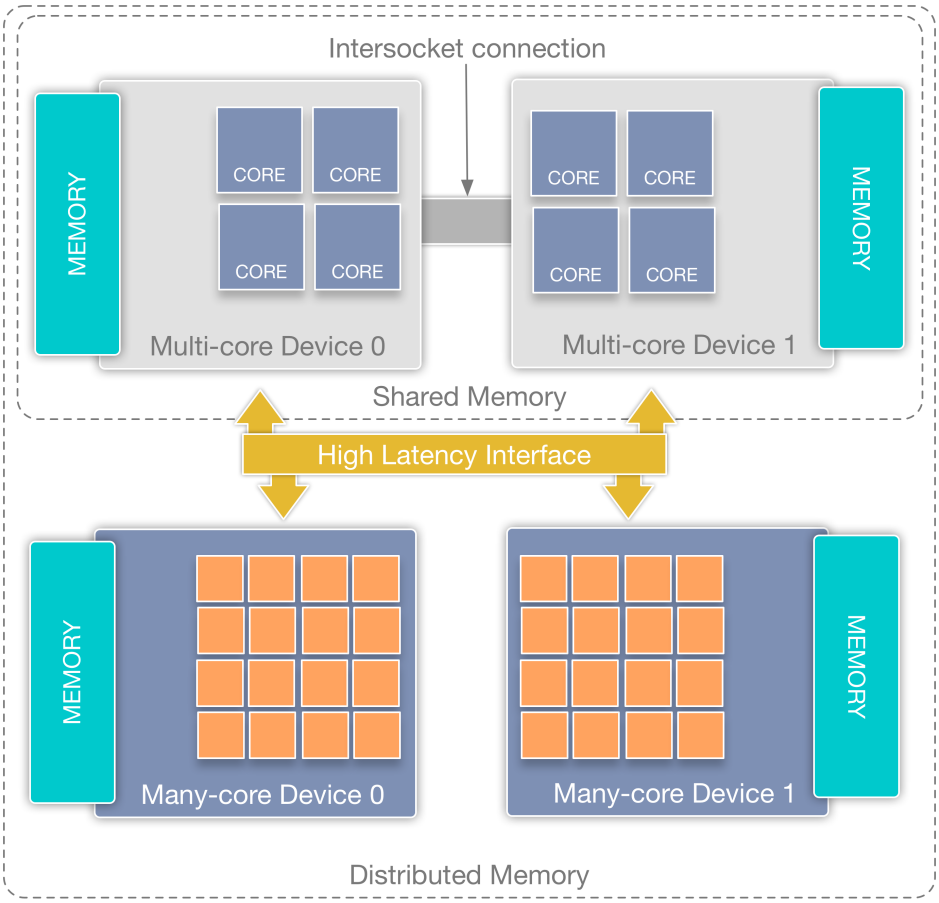


Figure 3: Representation of accelerators in distributed memory

2.3 HARDWARE

2.3.1 *CPU*

For several years the manufactures focused only on increasing the CPU's clock speed, which led the code developers to pay little attention on optimizing their code as it would run faster with each new hardware generation.

In 2005 the CPU's clock speed reached around 4 GHz and since then it became more challenging to increase it, due to the cost in terms of power consumption and heat dissipation. This forced the manufacturers to start exploring new forms to increase the throughput of their CPUs, so they began to increase the number of processing units, known as cores, towards multi-core and parallel computing.

Today's Intel Xeon multi-core devices go up to 22 physical cores capable of executing 44 hardware threads simultaneously (Figure 4). This improved parallelization of computations (doing multiple tasks at once).

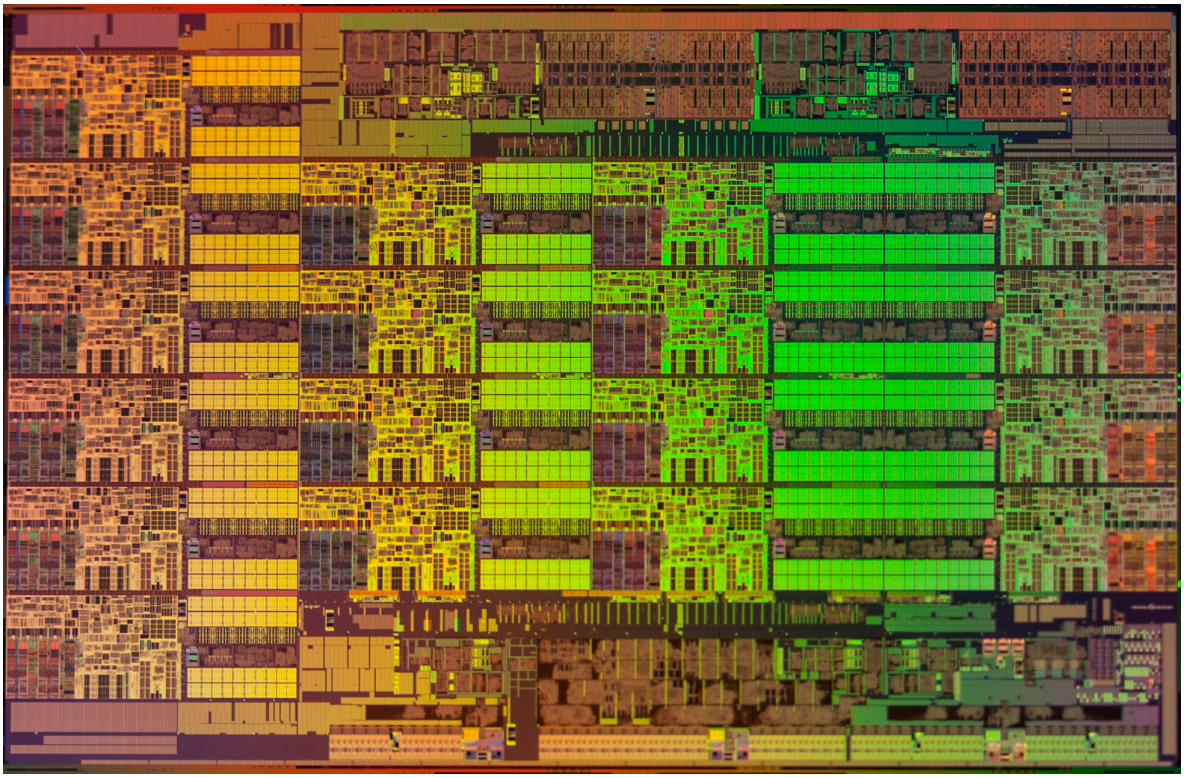


Figure 4: Intel Xeon Processor E5 v3 diagram (from: [8])

2.3. Hardware

This ended the "free lunch"[30] speedups that developers were accustomed to for many years, forcing them to parallelize code to achieve performance improvements in multi-core systems.

However, having a large number of cores does not necessarily result in an improvement in performance, since it depends not only on the software algorithms and their implementation but also on the percentage of concurrent data accesses. Possible gains are also limited by the fraction of the software that can not run in parallel on multiple cores, as expressed by the Amdahl's law[15].

$$S(p) = \frac{1}{f_s + \frac{f_p}{p}} \tag{1}$$

The theoretical speedup limit can be predicted using Amdahl's Law expression (1), where f_s represents the serial fraction of code, f_p the parallel fraction of code and p the number of processors. For example, if the parallel section of an application represents 90 percent of its run-time, using a 44 core processor, the speedup will be 8.3x at best.

The following chart (Figure 5) provides an even better notion of the impact that the percentage of code that can be parallelized has in the possible speedup of an application when we using multi-core processors.

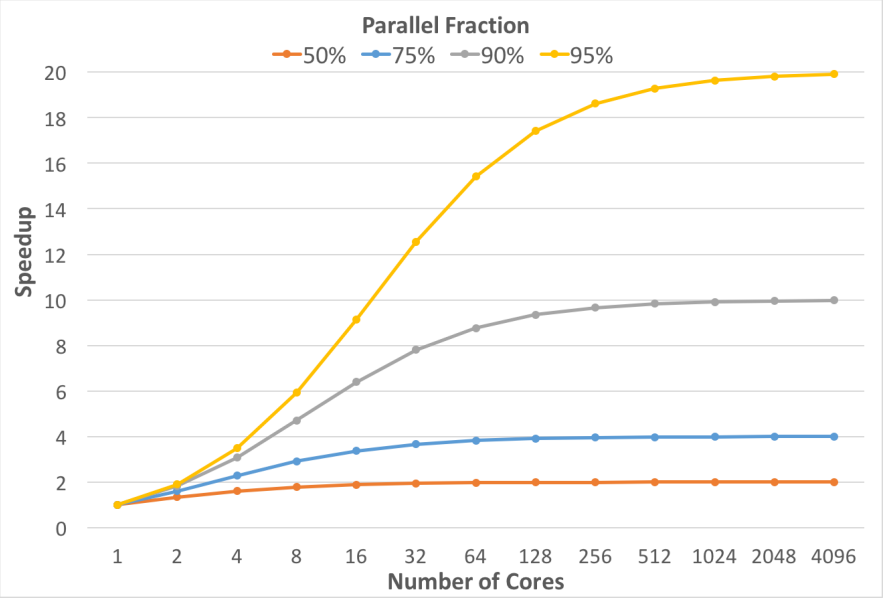


Figure 5: Amdahl's Law theoretical maximum speedup

In the best case, the so-called embarrassingly parallel problems, may reach speedup factors near the number of cores, or even more if the problem is split up enough to fit within each core's cache(s), avoiding the use of a much slower layer of the memory hierarchy.

The Amdahl's Law shows us that it is often impossible to make speedup scale linearly as we use more cores, due to sections of code that can not be parallelized and that often can not be worked around.

Gustafson's law[13] gives a less pessimistic and more realistic assessment of parallel performance and proposes that software developers tend to increase the size of problems (e.g. compute more scientific data) as the resources improve. Therefore, if faster equipment is available, larger problems can be solved.

2.3.2 *Graphics Processing Unit*

The GPUs are the most common accelerator devices since they are present on almost every desktop. CUDA[3][27] by Nvidia, which is a parallel computing platform and application programming interface (API) model, allows the software developer to exploit the ability to solve specific massively data parallel problems using the GPUs high computational power.

The Nvidia Tesla[18] products are devices that are meant specifically for high performance computing, so they lack of ability to output images to a display unlike consumer level (e.g. Geforce cards). These high performance computing devices are commonly used in simulations, in large scale calculations (especially floating-point calculations) and in high-end image generation for applications in professional and scientific fields.

The high performance computing GPUs do have some other characteristics that distinguishes them from general purpose GPUs, e.g they are designed to fit in cluster nodes, they come with different cooling options, they come equipped with a larger memory and even the chip itself is different, offering more processing units and larger memory caches.

Kepler[22] was Nvidia's first microarchitecture to also focus on energy efficiency unlike the previous architecture (Fermi) which was mainly focused on increasing performance. With Kepler, Nvidia abandoned shader clock and implemented a unified GPU clock, this not only simplified static scheduling of instruction but also resulted in a higher performance per Watt.

3 Compute Unified Device Architecture.

2.3. Hardware

According to Nvidia, two Kepler cores use approximately 90% power of one Fermi core, and the change to a unified GPU clock scheme delivered a reduction in power consumption of almost 50% .

Figure 6: NVIDIA Tesla (GK110) block diagram (from: [22])

Figure 6 shows the Kepler architecture organisation in two main components the Streaming Multiprocessors (SMX) and the internal memory hierarchy.

The SMX are complex processing units responsible for performing all computations on the GPU, and there may be up to 15 in a single chip. Each of the Kepler GK110 SMX units features 192 single-precision CUDA cores, each core has a fully pipelined floating-point and integer arithmetic logic units and they are able to deliver 3x more performance per Watt compared to the SM in Fermi. Kepler also added more single-precision/double-precision units per SMX.

Some other innovations are the Hyper-Q that allows multiple CPU cores to simultaneously launch different kernels on the same GPU, dynamic parallelism that allows to dynamically generate new workloads to process without the CPU interference and GPUDirect which reduces the communication latency by creating a direct connection to solid state drives and other similar devices, allowing multiple GPUs to share data without the interference of the CPU.

Figure 7: The Pascal GP100 Streaming Multiprocessor (from: [23])

The Nvidia Pascal GP100[23] (Figure: 7) has 3840 CUDA cores, 16GB of VRAM delivered in four HBM2 (High Bandwidth Memory) memory stacks, a 1 TB/s memory bandwidth and a memory bus width of 4096 bits. The HBM2 is built with layers of DRAM chips stacked into dense modules, with wide buses and it is located inside the same package with the GPU.

The GP100 increases performance not only by adding more SMs than previous GPUs, but by making each SM more efficient. Each GP100 SM has 32 FP64 units, providing a 2:1 ratio of single-precision/double-precision throughput. Compared to the 3:1 ratio in Kepler GK110 GPUs the Pascal GP100 is capable of processing FP64 workloads more efficiently.

Pascal is also the first Nvidia GPU to feature the company's new NV-Link[12] technology that Nvidia claims to be 5 to 12 times faster than PCIe 3.0, speeding up data transfers between the CPU-host memory and the GPU memory, overcoming a key bottleneck for accelerated computing today.

2.3.3 *Intel Many Integrated Core Architecture*

Intel has its own many integrated core line known as the Xeon Phi[8] that has key differences from the NVidia GPUs. The Intel Xeon Phi is composed by multiple interconnected Pentium Cores tha, thanks to the transistors size reduction, since the era of the Pentium, it is possible today to have 60 or more cores on one slab of silicon.

The high end model, from the first generation of the Intel Many Integrated Core devices, had 61 cores and 16 GB GDDR5 RAM and used Intel's Knights Corner architecture (Figure 8).

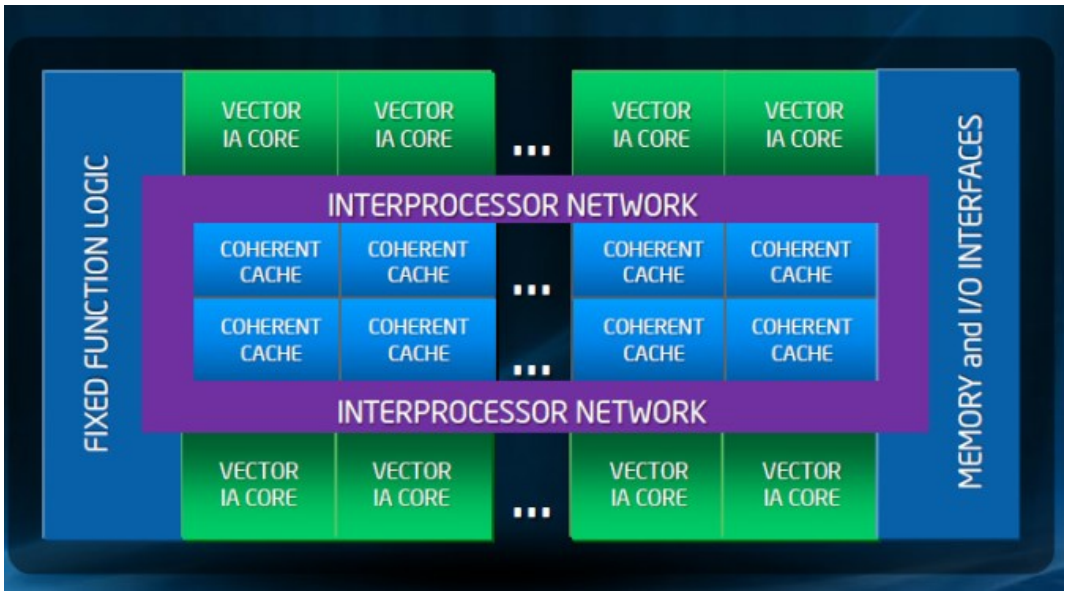


Figure 8: Representation of Intel Many Integrated Core architecture (from: *wiki.expertiza.ncsu.edu*)

Intel claimed that applications could be easily ported to run on the device, since the cores were based on a modified version of P54C design using the same instruction set as conventional x86 CPUs. Although, the code ported to Xeon Phi was normally inefficient out-of-the-box, requiring extra effort from the software developer to attempt to circumvent the inefficiencies.

Each core had the ability to run 4 threads simultaneously, and most of the massive parallelism was obtained by using the vectorisation capabilities provided by the available 32 vector registers 512-bit wide. However on Intel's Knights Corner architecture, only a small set of vector operations were implemented in the hardware and depended on the emulation, carried by the compiler, for the most complex ones.

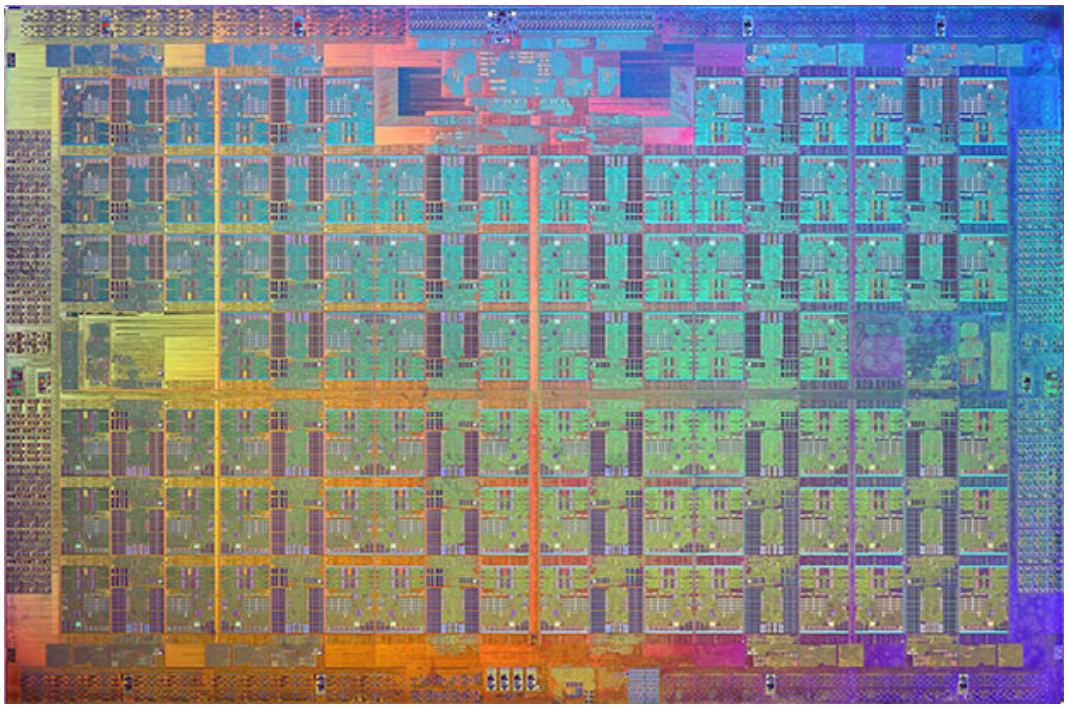


Figure 9: Intel Knights Landing Processor Die Map (from: [29])

Intel most recent version of Xeon Phi, also known as the Knights Landing[29], has a new chip with a 72-core processor solution manufactured on a 14nm process (Figure 9) structured as a set of 36 interconnected tiles, where each tile has a dual core structure with share L2 cache and each core has two vector processing units (512-bit wide, AVX-512). The new Knights Landing chip double-precision performance exceeds 3 teraflops and over 8 teraflops of single-precision performance. It also has 16GB of on-package MCDRAM[4] with a memory bandwidth of 500GB/s, which Intel claims to be five times more power efficient than GDDR5 and three times as dense.

Another big difference between the current and the previous version of Intel's Xeon Phi is that the the Knights Corner was a coprocessor, and it could not be used without a Xeon host, while the Knights Landing is only sold as a processor and not a coprocessor.

Knights Landing also comes with a new instruction set, more similar to x86, allowing an easier port of most C++ features, and will fully support Intel's Advanced Vector Extensions (Intel AVX-512).

4 MCDRAM is a high bandwidth ($\tilde{4}$x more than DDR4), low capacity memory, that can be configured as a third level cache or as a distinct NUMA node.

3

SCHEDULING DATA AND WORKLOADS

In a heterogeneous environment it is extremely complex to efficiently balance workloads among multiple devices that have different computational throughput for different types of tasks. Tasks that are highly parallelizable and/or memory intensive will in most cases perform better on a GPU device while compute intensive tasks should perform better on a Xeon Phi.

The distribution of the most suitable tasks to each device would potentially increase the computational throughput of the system, however in most cases it is impossible to anticipate the complexity of the tasks to exclusively send those that are ideal to each device.

Computational intensive tasks can also have irregular levels of intensities which increases the difficulty of efficiently distributing tasks among the the multiple devices due to their unpredictability and the differences on their execution time.

A static scheduler is far from being the best scheduling solution given that the multiple computing devices take different amounts of time to perform the same exact regular task, due to their different computational capabilities, let alone if tasks are irregular.

To solve this complex problem, several solutions address this issue in multiple ways. The simplest solution is by using dynamic a scheduler that distribute the same amount of work to each device as soon as they are become idle (known as the demand-driven[28] scheduling). This solution, despite being simple to implement, does not always lead to an optimal workload balancing, since it largely depends on the size of workload that is distributed among the devices. This size is defined by the user, and it may take several attempts before reaching the ideal workload size, and even after that, it may not be optimal for other work especially if the workloads have irregular computational intensities.

Another method is the adaptive scheduling. This method is more complex than the previous one and is also harder to implement, since the scheduler is capable to identify the performance level of each device and decide how much work each device receives. The main factor that is

usually taken to consideration, to identify the amount of work that each device receives, is the throughput of the device, i.e. the amount of work that each device is able to process in a certain time interval.

An example of a adaptive scheduler is the Qilin[19] scheduler. This scheduler requires that the first time that the applications runs, for an specific dataset, the user has to run it in sampling mode. In this first run, the scheduler tests the multiple devices performance level with different workload sizes and, in the end, it stores in a data base the percentage of workload that each device should receive the next time the application computes the same exact dataset. The main disadvantage of the Qilin scheduler,is that it requires that whenever the user changes the dataset, that he wishes to compute with the application, the first time that it computes is not expected to perform in the most efficient way, since it has to sample it.

A solution for all the previous scheduling issues is to use a scheduler which does not require that the user inputs any info regarding the size of the workload nor is it necessary to perform a sampling run. Ideally, the scheduler is able to in run-time, compute the optimum workload balance among multiple devices and is also able to react to possible performance breakage of one of the devices by quickly rebalancing the workloads.

3.1 Frameworks to Address Homogeneous Platforms

3.1.1 *POSIX Threads*

Initially each CPU manufacturer had their own proprietary versions of threads, and as it was expected, each version was quite different from one another making it virtually impossible for developers to create portable multithreaded applications.

So to be possible to fully exploit the capabilities provided by threads, it was necessary to develop a standard programming interface, which is known today as pThreads[21].

pThreads allows developers to spawn multiple threads that have different flows of work that execute at the same time (in parallel). But even so, the developers still had to worry about a few low-level details (e.g. as deadlocks), and therefore it was necessary to develop new libraries that would allow developers to work at a higher level of abstraction where they would not have to worry about such details.

3.1.2 *OpenMP*

One of the most popular high level libraries for parallel programming in homogeneous systems, the OpenMP[7] is an API that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran on most platforms processor architectures and operating systems. It uses a scalable and a portable model that delivers to the developers a clean and yet flexible interface to create parallel applications, even for most unskilled programmers.

Every OpenMP programs start as a single process known as the master thread. This thread executes sequentially until it encounters the first parallel region. When this happens, the master thread spawns a team of parallel threads and the program that is inside the parallel region is executed by this team. When the work is complete, each thread from the team synchronize and terminate, leaving only the master thread (Figure: 10).

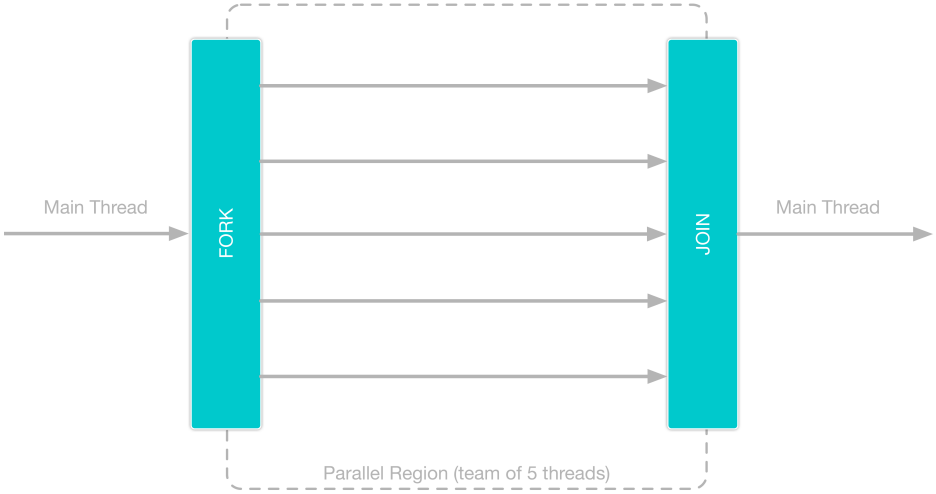


Figure 10: Representation of OpenMP work sharing strategy

By default, OpenMP schedules parallel loops statically where an equal number of iterations is given to each thread. OpenMP also has a built-in dynamic scheduler, which assigns one iteration to each thread and, when the thread finishes, the scheduler assigns to it the next iteration that hasn't been executed yet. The dynamic scheduler implemented in OpenMP is the best approach to follow, specially when dealing with iterations that might have irregular computational intensities.

The most recent versions of OpenMP offer the possibility to offload some work to GPU hardware accelerators through dedicated directives that mark the target regions. This version of OpenMP also controls data transfers between host and device memories (GPU) however; it does not offer an efficient load balance among CPU and accelerator simultaneously.

3.1.3 *Intel TBB*

Intel Threading Building Blocks[17][25] is a C++ template library developed by Intel for parallel programming on multi-core processors and it comes with a task scheduler that can efficiently balance workloads and also conduct some cache optimisations across multiple logical and physical cores. It offers thread-safe containers like `concurrent_vector` and `concurrent_queue` and generic parallel algorithms, like `parallel_for` and `parallel_reduce`.

An off-the-shelf scheduler with a work stealing strategy is implemented, where it employs a specified number of threads, called workers and each worker then maintains a local lock-free queue to store the tasks that were assigned to it. If a task queue becomes empty, its respective worker will attempt to steal a task from other busy workers at a synchronization point.

3.1.4 *Chapel*

Chapel[1][31] is a parallel programming language that is still being developed as part of the Cray Cascade project, a participant in DARPA's[2] High Productivity Computing Systems program, which is considered as a modern-day Manhattan Project[3].

This language follows a multithreaded parallel programming model at a high level by supporting abstractions on data parallelism, task parallelism, and nested parallelism. It can balance loads efficiently, and it also implements optimizations for data and computation locality.

Although Chapel may rival with OpenMP performance, it can be difficult to rival with MPI since programmers can implement tricks in order to make a more efficient communication system than the one that Chapel offers. On the other hand, Chapel has much better semantics for expressing a parallel algorithm than OpenMP or MPI. So, for expressing shared memory

1 Cascade High Productivity Language.
2 Defense Advanced Research Projects Agency.
3 Manhattan Project was a research and development project that produced the first nuclear weapons during World War II.

programs, Chapel could be good solution but not such a good one for distributed memory.

A negative aspect of Chapel, besides the fact that programmers have to learn a new language, it is that there are very few known libraries available (e.g. BLAS[4]) since these have to be ported.

Chapel may one day be able to fully support accelerators, and this will be something very interesting to analyze, but such a version is still under development.

3.2 Frameworks to Address Heterogeneous Platforms

3.2.1 *Cilk*

Cilk, Cilk++ and Cilk Plus[26] is a linguistic and run-time technology for multi-thread parallel C/C++ software development that provides both task and data parallelism constructs. Cilk was originally developed by the MIT Laboratory for Computer Science and after a few years it teamed up with Intel.

Intel Cilk provides three new keywords to the C/C++ syntax for reduction operations, and data parallel extensions. The keyword `cilk_spawn` can be applied to a function call, to indicate that it can be executed concurrentlye. The keyword `cilk_sync indicates` that execution has to wait until all spawns from the function have returned.

Using the keyword `cilk_for` instead of the regular `for` transforms a loop (without inter-loop dependencies) into a parallel loop indicating to the compiler that there is no ordering among the iterations of the loop.

Cilk scheduler uses a work-stealing strategy to divide procedure execution efficiently among multiple processors, and is applicable to multi-core and many-core programming.

4 Basic Linear Algebra Subprograms.

3.2.2 *OmpSs*

Developed in Barcelona Supercomputing Center (BSC), OmpSs[11] aim is to extend OpenMP with new directives to support asynchronous parallelism and heterogeneous platforms. Based on a task-oriented programming model similar to OpenMP, but in this case, all threads are created on start up and only one of them executes the main function. Every thread gets its work from a task pool and every thread is able to generate new work.

Expressing data-dependencies between tasks is possible on OmpSs using **in** (input), **out** (output) and **inout** (input/output) clauses. These clauses enables the ability for developers to easily specify which data a task is waiting for, and with this information OmpSs is able to generate a task dependency graph (Figure: 11) at run-time. Based on this graph, tasks are scheduled for execution as soon as all their predecessor has finished.

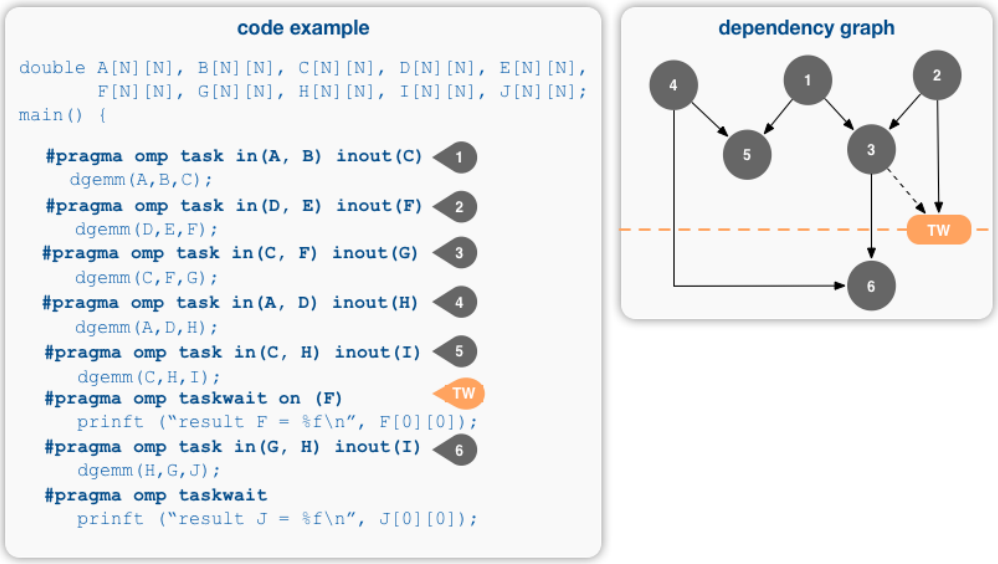


Figure 11: Representation of a OmpSs task dependency graph

OpmSs offers a target clause that gives the developer the ability to specify that a given element can be run in a set of devices (e.g. GPU). The target construct can be applied to either a task construct or a function definition.

To avoid communication overhead when working with GPUs, OmpSs performs regular `cudaFree` calls to prevent the device from entering in lower-power mode (deep C-state). OmpSs

also implements work stealing between GPUs, using `cudaMemcpyPeer` to copy data from their memories.

3.2.3 *Legion*

Targeted for users with extensive programming experience for heterogeneous platforms, the Legion[5] framework is a programming model that relies on an lengthy specification of the data structure of an application to be able to provide a high performance and efficient load balancing between the CPU and accelerators.

The framework uses logical regions to abstract the data handling from the users. This way, the developer can slice a region in order to increase the granularity of the tasks, enforce dependencies among tasks, and improve the scheduling and scalability of the algorithms.

During the execution, Legion adapts the workload of each computing device according to preceding execution time of the tasks on each device. In order to be possible for Legion to change the data chunk sizes, the developer needs to define a dicing function.

Legion can be used in a distributed memory, or in a shared memory environment. Currently it only supports NVidia and AMD GPUs hardware accelerators.

3.2.4 *StarPU*

StarPU[4] is a task programming library for heterogeneous architectures that offers a unified platform for task scheduling. The scheduler implements a Cilk like work stealing strategy where if one queue is empty then it can steal the work from other queues. The StarPU architecture implements various different kind of scheduling strategies (e.g. greedy and cost model guided scheduling model strategy).

The StarPU team claims that the architecture avoids unnecessary data transfers between accelerators, between accelerators and main memory since it only transfers data when requested. StarPU's scheduler is mainly focused on minimizing the cost of transfers between processing units and on using the data transfer cost prediction to improve the task scheduler decisions.

3.2.5 *Qilin*

Qilin[19] is a heterogeneous programming system that is capable of scheduling workloads among multiple heterogeneous devices. The authors claim that the scheduler operates best when it deals with regular workloads and, they admit that the scheduler will not have the best performance when dealing with irregular workloads.

Qilin scheduler saves in a database the execution-time projections, for all programs and datasets it has ever run, and when a program its run under Quilin for the first time, it runs in training mode.

When running in training mode, Qilin divides the entire workload in equal parts among the workers, each part is then divided into smaller parts with different sizes. Then the time that each worker takes to compute each smaller part is stored in a database and when all the devices finish processing their workloads, Qilin scheduler uses the stored run-times from each device and generates a linear equation, using the method of curve fitting, for each device. The next time that the same program runs with the same dataset under Qilin, the scheduler will intersect the multiple linear equations (one from each device) and will obtain the ideal workload size for each device.

The problems with this scheduler are that on the first run (training mode) will most certainly have a bad run-time. This scheduler will not work properly with irregular workloads (as it was admitted by the authors) and balancing is done only once (in the beginning) meaning that it cannot react to changes of performance of any worker during run-time.

4

SCIENTIFIC WORKLOADS

Scientific computing is a multidisciplinary field where non-computing scientists use advanced computing capabilities to solve complex problems.

Scientific code usually relates to modeling and simulation (in biology, physics, etc), numerical analysis, and quantitative data analysis. Due to the complex nature of these applications the code needs to properly use the available computing resources to process more data in less time using more complex algorithms in order to produce higher quality results.

Scientific workloads are typically irregular, mostly due to a filtering process that is usually present in scientific applications to discard data that, for some reason, have no interest for analysis. The data elements that are filtered and discarded, do not constitute a significant burden in computation, while those that pass the filtering process are subject to an extensive analysis which exponentially increases their computational weight. This usually is the main reason why an efficient scientific workload balancing among multiple devices is such a complex task.

4.1 THE CASE STUDY IN HIGH ENERGY PHYSICS

High Energy Physics (HEP) scientists at CERN often develop quantitative data analyses code to study and validate several HEP theories based on the analysis of the final state particles resultant from a collision at the Large Hadron Collider (LHC).

One of the main studies conducted by both ATLAS[1] and CMS[1][9] (the two largest research groups at CERN, each with their own particle detector at the LHC) relates to the research of the Higgs boson[2] in several decay channels.

1 Compact Muon Solenoid.

4.1. The Case Study In High Energy Physics

One of the most important decay channels is the production of the Higgs boson coupled to top quarks, known as $t\bar{t}H$ production. Two protons beans are accelerated in opposite directions at the LHC and collide at the core of a given detector, such as the ATLAS detector (Figure 12). A chain of decaying particles is created, from which only collisions that have the topology represented in the figure 13 should be considered for $t\bar{t}H$ studies.

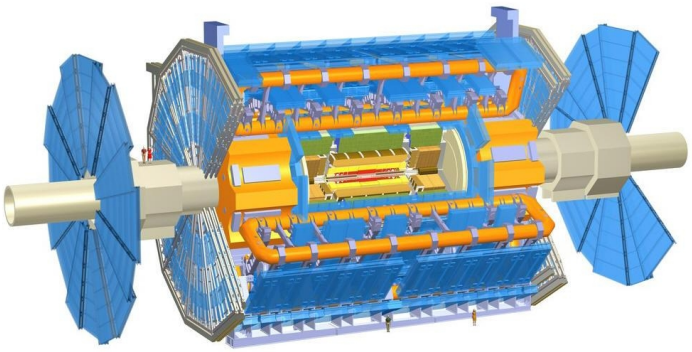


Figure 12: Computer generated cut-away view of the ATLAS detector (from: wikipedia)

The particle detector is only capable of measuring the bottom quarks, detected as a jet of particles, and leptons (muon and electron). Particle physicists developed code to separate collisions with different topologies and reconstruct both Higgs boson and the top quarks. Note that reconstructions characteristics are inferred by software through the conservation of momentum.

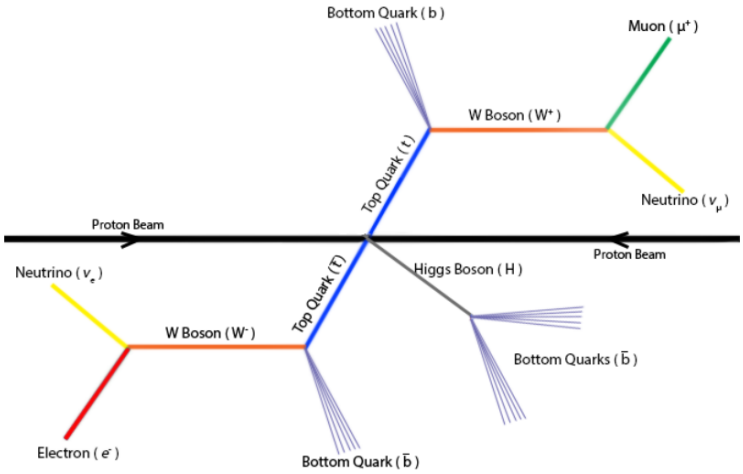


Figure 13: Representation of the $t\bar{t}H$ event (from: [24])

4.1. The Case Study In High Energy Physics

As shown in figure 13, four jets and two leptons are required in a $t\bar{t}H$ collision (know as event). Two of those jets, combined with two leptons are necessary to perform the reconstruction of the $t\bar{t}$ system, and the remaining two jets are used for the Higgs boson reconstruction. To analytically determine the characteristics of the neutrinos it is necessary to perform a kinematical reconstruction, where every possible combination of jets and leptons has to be tried and evaluated, and only the most accurate reconstruction of each event is considered. In this first phase, a $t\bar{t}$ system reconstruction is attempted. If it has a possible solution, the Higgs boson is reconstructed from the jets of the two remaining bottom quarks. The overall quality of the event processing depends on the quality of both reconstructions.

Particle detectors have an experimental resolution associated with every measurement of approximately $\pm 1\%$, which will have an impact on the reconstruction of the event and will affect the quality of the results. Physicists improved theirs reconstruction algorithms to perform an extensive search within the detector experimental resolution. This amount of reconstructions is performed for each event, where the input data is randomly varied within the resolution. The resolution that best fits the theoretical model is considered for the event, thus improving the result quality.

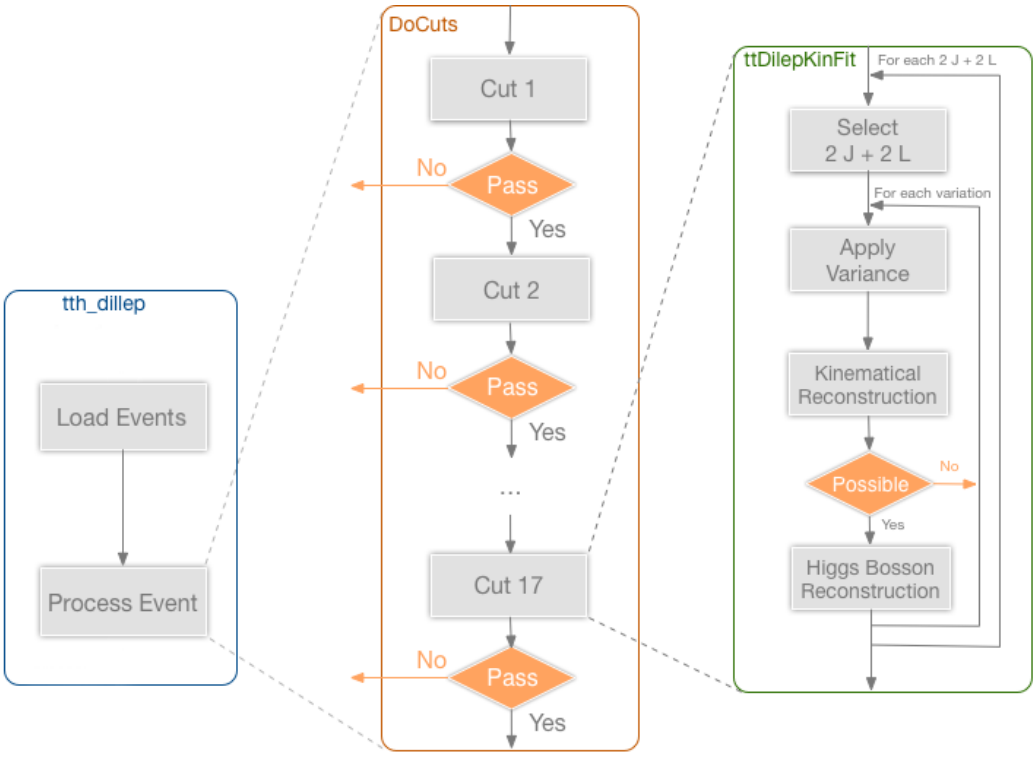


Figure 14: Representation of the ttH_dilep application work flow

4.1. The Case Study In High Energy Physics

One of the reconstruction software available is a scientific quantitative data analysis application was developed by ATLAS physicists to automate the reconstruction of events with the $t\bar{t}$ topology, known as ttH_dilep. This code, which structure is schematised in figure 14, loads the information of a set of events in an input data file to a structure in memory. Each event is submitted through a series of pipelined tasks, known as cuts in the particle physicists terminology, which purpose is to evaluate certain event characteristics to filter out events of different topologies. An event that passes all cuts is reconstructed in cut 17.

Each $t\bar{t}$ event information may contain up to 20 different jets and leptons associated, due to other decaying processes measured simultaneously by the detector. The reconstruction has to test every possible combination of 2 jets + 2 leptons to reconstruct the neutrinos, top quarks (kinematical reconstruction) and, if possible, the Higgs boson, from which only the best is kept. This process is repeated 1000 times, varying slightly the measured data, to overcome the detector experimental resolution.

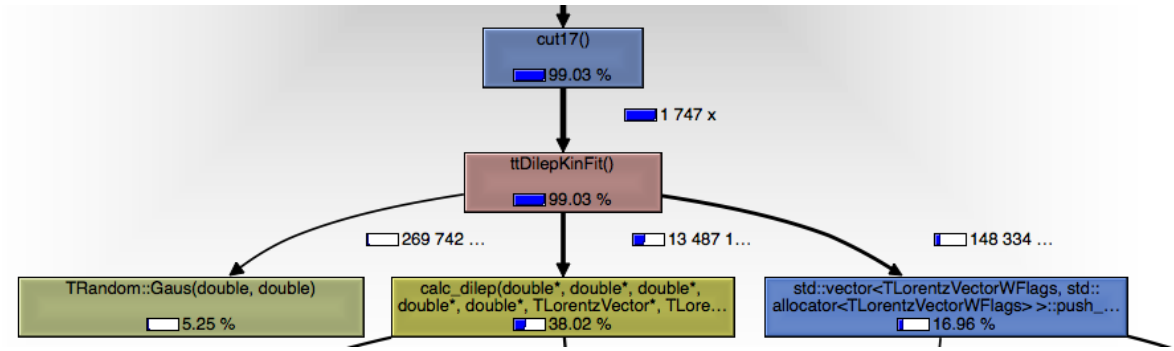


Figure 15: ttH_dillep callgraph (Zoomed in Cut17)

The ttH_dilep applies the same tasks pipeline to the independent events in the dataset. However, the execution time of each event pipeline is irregular due to the filtering cuts and reconstructions that sometimes are not possible. The execution time of the pipeline can vary up to 6 orders of magnitude, when compared to an event that is discarded by the first cut with an event that goes through all the cuts an is reconstructed (with 20 possible jets/lepton combinations). The cut 17 accounts for 99% of the overall application execution time (as it shows in figure 15), considering that less than 40% of the events were fit to be reconstructed.

The ttH_dilep has a simple parallelization scheme where the tasks pipeline is applied simultaneously to different events in the dataset, implemented using OpenMP. However, it was shown in [24] that this implementation did not scale properly when using more than 16 com-

puting units.

This may impact the speedup obtained by the proposed scheduler, since it does not balance the workload among each individual computing unit. The scheduler relies on the programmer to provide efficient parallelization strategies on each device, focusing on the workload balance among the devices.

4.2 HEP-Frame - A framework for computational science

The HEP-Frame is a user-centered Highly Efficient Pipelined Framework which aims to aid scientists, from any scientific field, in the development of sustainable parallel scientific applications with a flexible pipeline structure. The framework offers the computational scientist a user friendly environment to develop parallel applications, while automatically dealing with the parallelization complexities transparently to the user.

HEP-Frame provides a user-friendly interface that does not require any specific tuning for each individual scientific code or computing platform. Optimization efforts focus on improving both the execution throughput of an input dataset and the execution time of each element in the dataset, in offline quantitative analyses.

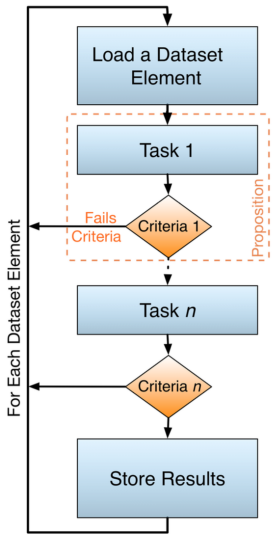


Figure 16: Structure of a typical flexible pipelined application (from: [24])

Pipelined applications typically have the structure presented in figure 16. For the HEP-Frame a proposition is a computational task, which is applied to a dataset element, and an evaluation of a given dataset element characteristic that may restrict the execution of subsequent propositions. The tasks may be computationally simple or complex, and the criteria may discard different amounts of dataset elements.

HEP-Frame scheduler parallelizes the execution of the pipeline to different dataset elements, while implementing the ability to analyze the level of performance of each task at run-time and explore new pipeline orders, seeking the one that has a higher level of computational efficiency, but always ensuring that the dependencies between tasks (defined by the developer) are kept.

The ttH_dilep, which is the case study used in this thesis, is actually the prototype used by HEP-Frame to validate its functionality. The case study was implemented using the HEP-Frame and so the scheduler featured in this thesis was implemented in HEP-Frame giving the possibility to use the scheduler in other scientific applications that use this framework. However, the ttH_dilep code had to be manually ported to operate with Intel Xeon Phi, and multiple auxiliary data-structures had to be implemented to be able to offload the $t\bar{t}H$ events to the device, as they events were originally allocated in data-structures that were incompatible with the Intel Xeon Phi offload mechanisms.

5

AN ADAPTIVE SCHEDULER FOR AN HETEROGENEOUS SERVER

To dynamically schedule embarrassingly parallel applications, among devices with different architectures and performance level, developers tend to implement a scheduler that follows a demand-driven approach. In this approach, when a worker (device) is free, or about to finish its previously assigned workload, it requests a new workload from the task stack that have not yet been processed. The workload size is set by the user and an ill-fitting size may result in a loss of performance. If the user set a size too small, this will increase the number of communications made between the various devices, which will increase the weight of the scheduler. On the other hand, if the size is too large, this may cause a significant loss of resource use, since it may occur the situation where a device receives the last workload from the stack and immediately after one, or all the other devices, become free (finished their previously assigned workload) and are forced to wait for the other device to finish its large workload.

To overcome these limitations a refinement of the demand-driven approach is presented, which is tuned for each device characteristics, with an adaptive time interval between consecutive workload requests. The goal is to maintain the devices busy as long as possible, adapting each device workload to minimize the amount of idle time.

The scheduler parallelization strategy assigns a task to execute at each computing unit at each given time, while the amount of allocated tasks to each device should be one order of magnitude higher than the number of available logical cores of the device with the higher number of cores. It considers each multi-core/many-core device as a distinct worker with its own performance index, which is updated throughout the application execution time. The inexistence of dependencies in the dataset allows the scheduler to be simpler and lightweight.

The amount of work assigned to each device (or worker) is iteratively refined based on their performance. The scheduler starts with a subset of the workload (with a predefined size) and allocates equal sized workloads for each of the workers. In the next iteration the workload size is adapted according to the performance of each worker: workers with more computational

throughput will be assigned larger workloads.

The scheduler measures the amount of time that each worker is idle waiting for others to complete their portion of the load, and combines that with the size of the workload that was given to each worker to compute their performance index (2). Figure 17 shows a load balancing example of the scheduler.

The expression (2) computes the new workload size based on the idle time and the previous workload size.

$$WrkLp_i^{n+1} = \frac{WrkLp_i^n + (WrkWp_i^n \times WrkLp_i^n)}{\sum_{p=0}^{d} WrkLp_p^n + (WrkWp_p^n \times WrkLp_p^n)} \tag{2}$$

Where n is the number of the scheduling iteration, $WrkLp$ represents the worker load percentage, $WrkWp$ represents the percentage of time that the worker remained idle waiting for others to complete their portion of the load, i and p represents the device ID and d is the total number of devices.

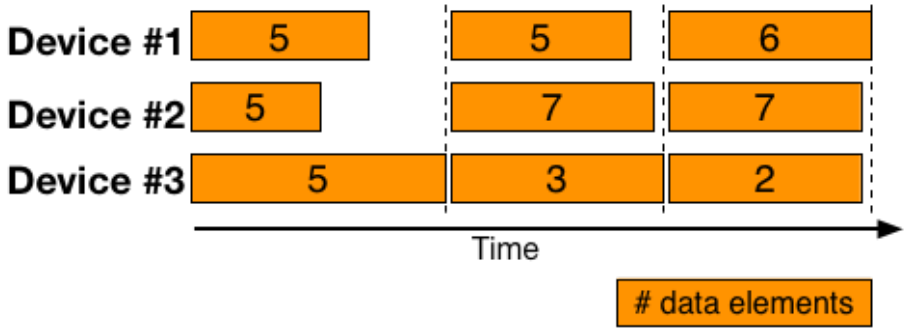


Figure 17: The proposed scheduler balancing sets of 15 data elements among multi-core and many-core devices in two checkpoints

When the workload size stabilizes between two iterations, where the difference between the previous and new sizes is less than a given threshold (expression (3)), the scheduler doubles the size of the subset of the data that is divided into chunks. This helps reduce the scheduler overhead and the amount of memory transfers passing through the PCI-Express interface. If the scheduler calculates a new workload size with a variation higher than the threshold, the size of the next data subset to be distributed is halved, ensuring that the scheduler efficiently

adapts to irregularities in the workload pattern.

$$\sum \left(\left| {WrkLp_{cpu}}^{n+1} - {WrkLp_{cpu}}^{n} \right| + \left| {WrkLp_{mic}}^{n+1} - {WrkLp_{mic}}^{n} \right| \right) \tag{3}$$

The data subset to distribute into workloads can be increased/decreased by a factor of 2 for up to 4 times. Increasing its size more than 4 times is not beneficial to the performance, as adapting the workload to spikes in the tasks irregularity would take longer. More complex strategies could be used but the increased overhead would significantly impact the performance for faster applications.

The current implementation of the scheduler relies on the parallel code developed by the user for each device, while it performs a naïve parallelization in the Intel Xeon Phi, by processing different tasks simultaneously. The memory transfers were implemented using the Intel Compiler pragma directives.

The scientific application, used as case study to validate the scheduler, has a scalability limitations that caused a significant performance break and prevented from achieving better results when using two multi-core devices.

After profiling the scientific application with VTune[3], it was evident that the generation of PRN[1] represented a considerable part of the application's run-time and that its weight nearly doubled when going from one to two multi-core devices. As each thread was dedicating part of its run-time to generate multiple PRN using its own seed, it was considered that it would be more efficient to have a dedicated thread to generate a list of PRN.

This is a typical producer-consumer problem, so the solution for the PRN hotspot was to create an observer thread that would generate multiple pools of PRN and then feed them to the threads running ttDilepKinFit as soon as they need them to apply the variations in the reconstruction of the event. The observer thread manages the pools of PRN (generate new pools and free depleted ones).

1 Pseudorandom Number.

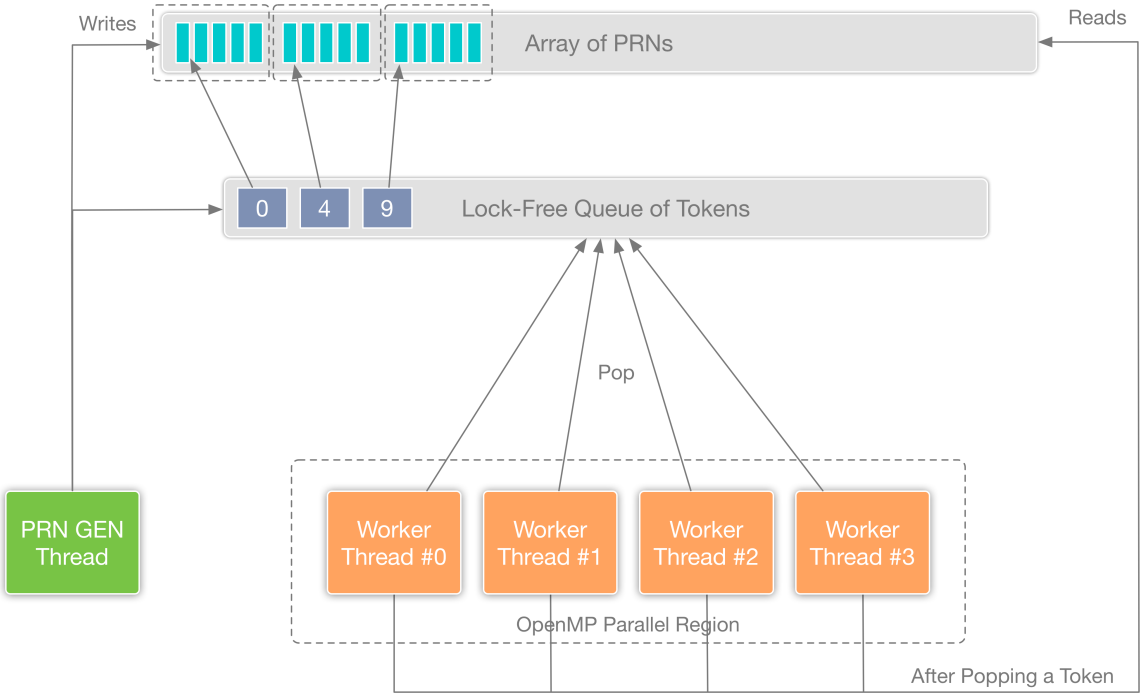


Figure 18: Visual representation of the PRN optimization

The access to the pool of PRN had to be performed simultaneously by the multiple threads without having to use locks, otherwise we would be creating a new hotspot caused by the fact that the threads would have to block themselves and wait for their turn to acquire the lock.

To solve this issue, a lock-free queue from Boost[10] C++ library[2], allocates a list of tokens that represent the positions from where each thread should start feeding (figure: 18). This way a block of PRN from the pool are reserved to a certain thread, guaranteeing that no other thread uses the same PRN.

2 Set of libraries for the C++ programming language that provide support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing.

6

EXPERIMENTAL RESULTS AND DISCUSSION

6.1 THE SERVER AND MEASUREMENTS CONFIGURATION

The main goal of the new scheduler was to minimize the waiting time of the available devices, while keeping the overhead as low as possible. The experimental measurements addressed the execution times of the scientific application that is used as case study, while using the scheduler to distribute workloads among the available devices, and the percentage of time that each device (or set of devices in a shared memory environment) remains idle. With these values we can compute how long the fastest device is waiting for the slowest, for each load distribution, and consider that as the scheduler performance (lower idle time means better scheduler performance).

A dual socket server was used with 12-core Intel Xeon 2695v2 CPU devices @2.4 GHz (2-way simultaneous multithreading) coupled with two PCIe cards, each with a 61-core Intel Xeon Phi 7120 @1.2 GHz (4-way simultaneous multithreading). Four different heterogeneous server configurations were used as testbed, with the following computing devices: (i) 1 multi-core + 1 many-core; (ii) 2 multi-cores + 1 many-core; (iii) 1 multi-core + 2 many-cores; and (iv) 2 multi-cores + 2 many-cores. The application was always configured to use 2 threads per CPU core and 4 threads per Xeon Phi core (1 thread per logical core). The granularity of the initial workload to distribute was tuned so that there are 10 times more independent tasks than threads on the Xeon Phi (the device with the most logical cores, as described in chapter 5) so that it is easier to explore the performance potential of this device.

To measure the execution times of the selected application case study, the ttH_dilep, a k-best measurement heuristic was used to ensure that the best results can be replicated, with k = 3 within a tolerance factor of 5% with a minimum of 6 measurements and a dataset with $\pm 750,000$ events.

We also present a table with speedups of hardware configurations with 1 or 2 Phi related to the same code being executed in only 1 multi-core or 2 multi-core devices. Note that this case study does not scale well when moving from 1 multi-core to 2 multi-core devices, nor the Xeon phi devices were adequately parallelized. These performance limitations are solely due to the inefficient parallel code running one each device, an issue out of the scope of this scheduler.

6.2 VALIDATION AND DISCUSSION

A strong point of this scheduler is its very low overhead. In the worst case measured, the scheduler had an overhead of about 1.68 milliseconds, which was approximately 0.000175% of the total application run-time.

To ensure that the scheduler generates a low overhead, the chunk size is doubled amount, when the difference between the workload sizes is less than a given threshold, reducing the number of memory transfers passing through the PCI-Express interface. Each time the chunk size is doubled, the duration of the next iteration also nearly doubles. Therefore, figure 19 should be seen carefully, since all iterations do not have the same execution time.

As for the key goal, to minimize the waiting times of the devices, figure 20 show the percentage of time that all workers were busy computing in parallel and the percentage of time that at least one of workers stayed idle waiting for the others to finish computing their workload. In other words, the idle time represents the percentage of time that the fastest worker spent waiting for the slowest in each iteration.

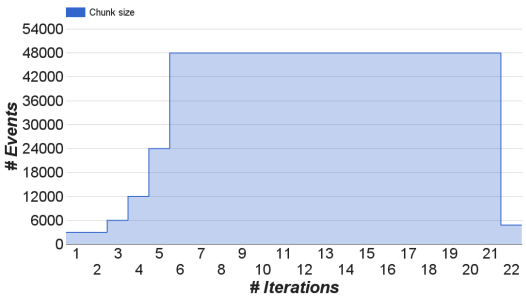


Figure 19: Set of events chunk size

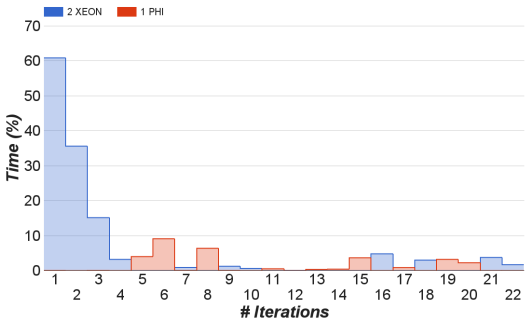


Figure 20: Percentage of waiting time

As we can observe from figure 20 the waiting time of all the workers decreases rapidly in the first iterations of the scheduler, stabilizing in around 3%. In other words both workers

spend 97%, of the run-time, working concurrently. Full concurrent efficiency usage happens when all workers are operating in parallel without any of them waiting for the others to finish computing their workload.

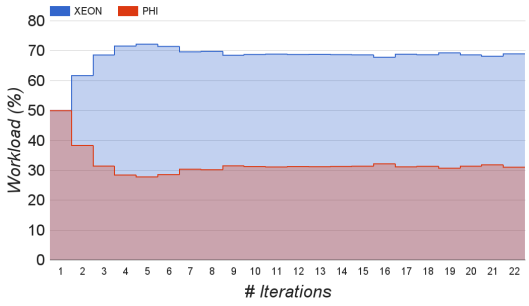


Figure 21: Workload balance by iteration

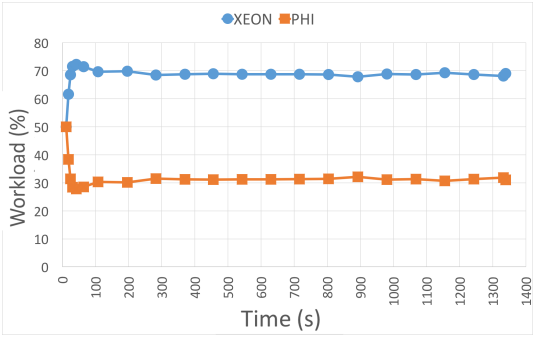


Figure 22: Workload balance by time

The figures 21 and 22 show how scheduler balances the workload throughout the multiple iterations and time. Figure 22 evidences how fast the workloads are stabilized. The system configuration that was used in these results was with 2 multi-core and 1 many-core.

The workload is virtually balanced in the first two iterations and these two iterations took only 14 seconds, which represents 1.04% of the total run-time.

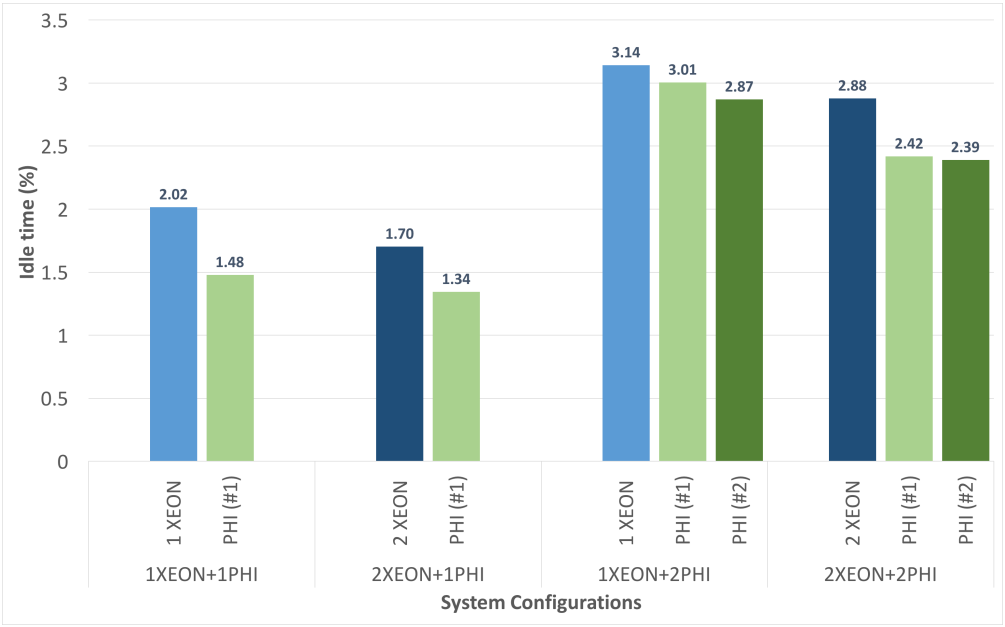


Figure 23: Percentage of waiting time of each device on every system configuration

6.2. Validation and Discussion

Figure 23 show the percentage of time that each device had to remain idle while waiting for others to complete their portion of the load, on every system configuration. Additionally, we can infer that in the worst case, the multiple devices had an occupancy rate of 97%.

However the scientific application contained a scalability limitations that caused a significant performance break and prevented from achieving better results when using two multi-core devices, as we can conclude by observing table 1.

Configuration	multi-core	Heterogenous	Speedup
1XEON	22m 32s	-	1 x
1XEON+1PHI	-	19m 46s	1,14 x
1XEON+2PHI	-	15m 59s	1,40 x
2XEON	33m 7s	-	1 x
2XEON+1PHI	-	24m 12s	1,37 x
2XEON+2PHI	-	18m 6s	1,83 x

Table 1: Speedup achieved on the multiple system configurations

In an attempt to identify what was causing the drop in performance when using both multi-core devices, a performance analysis was performed using the VTune.

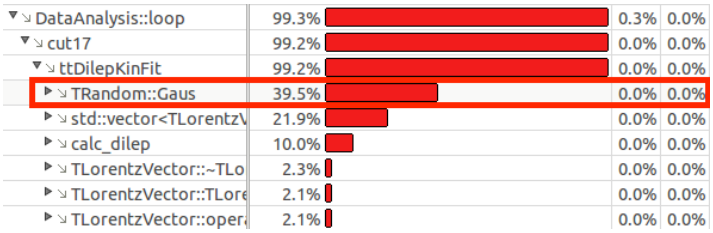


Figure 24: VTune performance analysis while using a single multi-core device

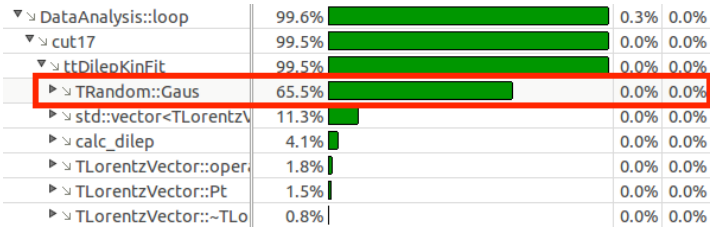


Figure 25: VTune performance analysis while using two multi-core devices

6.2. Validation and Discussion

With the help from these results (figures 24 and 25), it was evident that the generation of PRN represented a considerable part of the application's run-time and that its weight nearly doubled when going from one to two multi-core devices.

After implementing the PRN generation optimizations, the scientific application still had a scalability limitations that prevented from achieving better results when using two multi-core devices. In any case, an improvement in performance was obtained (Table: 2).

Configuration	multi-core	Heterogenous	Speedup
1XEON	10m 46s	-	2.01 x
1XEON+1PHI	-	9m 35s	2,11 x
1XEON+2PHI	-	8m 43s	1,83 x
2XEON	15m 18s	-	2,16 x
2XEON+1PHI	-	11m 30s	2,1 x
2XEON+2PHI	-	11m 28s	1,57 x

Table 2: Speedups on the multiple system configurations after PRN optimization
Note: Speedups are related to the run-times shown in Table: 1

Unfortunately, this optimization did not remove the limitations of the parallel code version for the multi-core device. We continue to observe a significant loss of performance when we use two multi-cores. Nonetheless, a significant improvement in performance was achieved across all test environments, managing to reduce the run-times to practically half from those presented in Table 1.

By improving the performance of the parallel version of the code used in the multi-core device, the scheduler ended up increasing the workload delivered to this device, reducing the workload delivered to the many-core devices. For this reason, as we can observe in Table 2, the use of one and two many-core devices generated a smaller increase in performance since these devices processed a smaller amount of data.

If the parallel version of the code used in the many-core is optimized, and its throughput of events increases, the execution time may potentially improve since the scheduler would distribute a larger workload to these devices.

7

CONCLUSIONS AND FUTURE WORK

This dissertation presented a model and the respective implementation of a dynamic adaptive scheduler capable of efficiently distributing irregular workloads on an heterogeneous server, across multiple devices (multi-core and many-core) with a very low overhead. This new scheduler is generic to deal with any type of work or device, and any number of devices.

The scheduler was implemented in the HEP-Frame so that it could potentially be used to aid scientists, from multiple scientific fields, and improve the performance of their applications on heterogeneous platforms.

The current version of the scheduler implements an efficient load balancing of irregular workloads with a very small overhead (less than 0.00018% of the total application run-time) on a heterogeneous platform that uses as an accelerator, one or multiple Intel Xeon Phi. It is also expected that the scheduler obtains such results in other heterogeneous server configurations.

The main reason why the scheduler produces such small overhead and that in its the worst case the multiple devices had an occupancy rate of 97% is due to its new heuristic that is lightweight and at the same time it is also very efficient, allowing the scheduler to make the right decision and send to each device the amount of work that close to the ideal in practically all its iterations. This heuristic also allows workloads to converge to the optimal size very early in the application execution, allowing a better exploitation of the computational resources of an heterogeneous server.

This scheduler has not been tested with a GPU device, because till date there is still no version of the `ttH_dillep` application compatible with this type of platforms. However, it is expected that the scheduler deals efficiently with GPUs, given that it operates regardless of the characteristics of the devices.

The parallelization of the code used inside the multi-core and many-core devices is not the responsibility of the scheduler, and it depends entirely on the code that the developer produces to properly scale as the number of cores increases.

A possible optimization of the scheduler is to handle the multi-core devices (Xeon) as separate devices in order to specifically offload workload to each multi-core device and reduce NUMA accesses where the memory access time depends on the memory location relative to the device where the code is being executed.

BIBLIOGRAPHY

[1] Aad, Georges, E. Abat, J. Abdallah, A. A. Abdelalim, A. Abdesselam, O. Abdinov, B. A. Abi et al. *The ATLAS experiment at the CERN large hadron collider.* Journal of Instrumentation 3, no. S08003, 2008.

[2] Aad, Georges, T. Abajyan, B. Abbott, J. Abdallah, S. Abdel Khalek, A. A. Abdelalim, O. Abdinov et al. *Observation of a new particle in the search for the Standard Model Higgs boson with the ATLAS detector at the LHC.* Physics Letters B 716, no. 1, pp. 1-29, 2012.

[3] Adhianto, Laksono, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John MellorCrummey, and Nathan R. Tallent. *HPCToolkit: Tools for performance analysis of optimized parallel programs.* Concurrency and Computation: Practice and Experience 22, no. 6, pp. 685-701, 2010.

[4] Augonnet, Cédric, Samuel Thibault, Raymond Namyst, and PierreAndré Wacrenier. *StarPU: a unified platform for task scheduling on heterogeneous multi-core architectures.* Concurrency and Computation: Practice and Experience 23, no. 2, pp. 187-198, 2011.

[5] Bauer, Michael, Sean Treichler, Elliott Slaughter, and Alex Aiken. *Legion: expressing locality and independence with logical regions.* In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, p. 66. IEEE Computer Society Press, 2012.

[6] Bryant, Randal, O'Hallaron David Richard, and O'Hallaron David Richard. *Computer systems: a programmer's perspective.* Vol. 281. Upper Saddle River: Prentice Hall, 2003.

[7] Chapman, Barbara, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming.* Journal of Computer Science Technology Vol. 10. MIT press, 2008.

[8] Chrysos, George. *Intel Xeon Phi^{TM} Coprocessor-the Architecture.* Intel Whitepaper, 2014.

[9] Collaboration, C. M. S., and R. Adolphi. *The CMS experiment at the CERN LHC.* Journal of Instrumentation 3, no. 08, 2008.

[10] Demming, Robert, and Daniel J. Duffy. *Introduction to the Boost C++ Libraries; Volume I-Foundations.* Datasim Education BV, 2010.

[11] Duran, Alejandro, Eduard Ayguadé, Rosa M. Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. *Ompss: a proposal for programming heterogeneous multi-core architectures.* Parallel Processing Letters 21, no. 2, pp. 173-193, 2011.

[12] Foley, Denis. *NVLink, Pascal and Stacked Memory: Feeding the Appetite for Big Data.* Nvidia.com (2014).

[13] Gustafson, John L. *Reevaluating Amdahl's law.* Communications of the ACM 31, no. 5, 1988.

[14] Hackenberg, Daniel, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. *An energy efficiency feature survey of the intel haswell processor.* In Parallel and Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International, pp. 896-904. IEEE, 2015.

[15] Hill, Mark D. and Michael R. Marty. *Amdahl's law in the multi-core era.* IEEE Computer, vol. 41, pp. 33-38, 2008. 2008

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

[17] Kukanov, Alexey, and Michael J. Voss. *The Foundations for Scalable Multi-core Software in Intel Threading Building Blocks.* Intel Technology Journal 11, no. 4, 2007.

[18] Lindholm, Erik, John Nickolls, Stuart Oberman, and John Montrym. *NVIDIA Tesla: A unified graphics and computing architecture.* IEEE Micro 28, no. 2, pp. 39-55, 2008.

[19] Luk, Chi-Keung, Sunpyo Hong, and Hyesoon Kim. *Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping.* In 42nd Annual IEEE/ACM International Symposium on Microarchitecture, pp. 45-55. IEEE, 2009.

[20] Martineau, Matt, Simon McIntosh-Smith, and Wayne Gaudin. *Evaluating OpenMP 4.0's Effectiveness as a Heterogeneous Parallel Programming Model.* In Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International, pp. 338-347. IEEE, 2016.

[21] Nichols, Bradford, Dick Buttlar and Jacqueline Farrell. *Pthreads programming: A POSIX standard for better multiprocessing.* O'Reilly Media, Inc., 1996.

[22] NVIDIA. *NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110 - The Fastest, Most Efficient HPC Architecture Ever Built.* NVidia Whitepaper, 2013.

[23] NVIDIA. *NVIDIA Tesla P100 - The Most Advanced Datacenter Accelerator Ever Built Featuring Pascal GP100, the World's Fastest GPU.* NVidia Whitepaper, 2016.

[24] Pereira, Andre, Antonio Onofre, and Alberto Proenca. *HEP-Frame: A Software Engineered Framework to Aid the Development and Efficient multi-core Execution of Scientific Code.* In 2015 International Conference on Computational Science and Computational Intelligence (CSCI), pp. 615-620. IEEE, 2015.

[25] Reinders, James. *Intel threading building blocks: outfitting C++ for multi-core processor parallelism.* O'Reilly Media, Inc., 2007.

[26] Robison, Arch D.. *Composable Parallel Patterns with Intel Cilk Plus.* Computing in Science and Engineering 15, no. 2, pp. 66-71, 2013.

[27] Sanders, Jason, and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming.* Addison-Wesley Professional, 2010.

[28] Shivaratri, Niranjan G., Phillip Krueger, and Mukesh Singhal. *Load distributing for locally distributed systems.* Computer 25, no. 12, pp. 33-44, 1992.

[29] Sodani, Avinash. *Knights landing (KNL): 2nd Generation Intel Xeon Phi processor.* In Hot Chips 27 Symposium (HCS), pp. 1-24. IEEE, 2015.

[30] Sutter, Herb. *The free lunch is over: A fundamental turn toward concurrency in software.* Dr. Dobb's Journal 30, no. 3, 2005.

[31] Weiland, Michele. *Chapel, Fortress and X10: novel languages for HPC.* EPCC, The University of Edinburgh, Tech. Rep. HPCxTR0706, 2007.