universität wien

DISSERTATION / DOCTORAL THESIS

Titel der Dissertation / Title of the Doctoral Thesis

Efficient Execution of High Performance Computing Applications on Asymmetric Heterogeneous Hardware Architectures

verfasst von / submitted by

Valon Raça

angestrebter akademischer Grad / in partial fulfillment of the requirements for the degree of

Doktor der technischen Wissenschaften (Dr. techn.)

Wien, 2019/ Vienna, 2019

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	UA 786 880
Studienrichtung lt. Studienblatt / field of study as it appears on the student record sheet:	Informatik
Betreuer / Supervisor:	ao. Univ.-Prof. Dipl.-Ing. Dr. Eduard Mehofer

Abstract

Heterogeneous systems are ubiquitous nowadays in the field of High Performance Computing (HPC). Heterogeneous devices suitable for parallel computing such as graphical processing units (GPU), x86 many core processors (Xeon Phi), accelerated processing units (APU), reconfigurable computing units (FPGA) and other types of specialized co-processors have been shown to be better performing or more energy efficient than CPUs for many types of data parallel applications. Usually, these devices are unevenly distributed over cluster nodes leading to asymmetric cluster configurations - especially in computational labs.

Programming heterogeneous clusters is a major challenge. Emergence of the OpenCL standard for heterogeneous computing has alleviated the problem of porting applications to different heterogeneous devices. But, OpenCL alone is not sufficient. Other parallel and concurrent programming paradigms have to be used in conjunction with OpenCL in order to enable efficient distribution of work over cluster nodes. Moreover, different behavior of the individual compute devices with respect to execution time and energy consumption has to be taken into account in order to meet the demands of the user for higher performance and energy-efficient computing.

The aim of this thesis is to deal with the problem of heterogeneous computing in cluster systems by providing a framework which supports partitioning and distribution of parallel programs into cluster nodes and by guiding the execution process to achieve better performance while lowering energy consumption. In particular, the results of this thesis are:

(i) *A Workload Partitioning Approach* which partitions the applications into medium-grain same-size work chunks and enables efficient distribution of work in heterogeneous asymmetric clusters while ensuring easier handling of device failures and poor performance,

(ii) *A Heterogeneous Cluster Computing Framework* - ***clusterCL*** - which supports our partitioning approach by providing an abstraction layer to the programmer and hiding the intricacies of programming asymmetric heterogeneous clusters,

(iii) *A Set of Optimal Algorithms for Work Distribution* that take into account execution time and energy consumption in a Time-Energy solution space and steer the execution process towards high performance, energy-efficiency or reasonable trade-offs between execution time and energy consumption,

(iv) *Evaluation of the Validity and Efficiency of our Approach* based on experiments with eleven HPC applications and two synthetic parallel programs from different scientific domains and benchmarks which are executed in three different mid-size asymmetric cluster systems with numerous heterogeneous devices of different types.

Zusammenfassung

Heterogene Systeme sind heutzutage im Bereich High Performance Computing (HPC) weit verbreitet. Heterogene Computing Devices wie Grafikprozessoren (GPU), x86 Manycore-Prozessoren (Xeon Phi), Accelerated Processing Units (APU), rekonfigurierbare Recheneinheiten (FPGA) und andere Arten von spezialisierten Co-Prozessoren haben sich für Parallelrechner und unterschiedliche daten-parallele Anwendungen als leistungsfähiger oder energieeffizienter erwiesen als CPUs. Meist sind diese Computing Devices nicht gleichmäßig über die Clusterknoten verteilt, wodurch asymmetrische Clusterkonfigurationen entstehen - vor allem in kleineren Computer-Labs.

Die Programmierung heterogener Cluster ist eine große Herausforderung. Die Verbreitung von OpenCL für heterogenes Computing hat das Problem der Portierung von Anwendungen auf verschiedene heterogene Devices erleichtert. Aber OpenCL allein reicht nicht. Zusätzlich müssen parallele und nebenläufige Programmierparadigmen in Verbindung mit OpenCL angewendet werden, um eine effiziente Verteilung der Arbeit auf die Clusterknoten zu erreichen. Weiters müssen unterschiedliche Charakteristiken bezüglich Ausführungszeit und Energieverbrauch der einzelnen Computing Devices berücksichtigt werden, um Leistungs- und Energieeffizienz-Anforderungen gerecht zu werden.

Ziel dieser Arbeit ist es, heterogenes Rechnen in Clustersystemen zu erleichtern und ein Framework anzubieten, das Partitionierung und Verteilung paralleler Anwendungen auf Clusterknoten unterstützt und durch Steuerung des Ausführungsprozesses bessere Laufzeit bei gleichzeitig niedrigerem Energieverbrauch erreicht. Die Ergebnisse dieser Arbeit sind insbesondere:

(i) *ein Workload Partitioning Ansatz*, der die Anwendungen in mittlere, gleichgroße Arbeitspakete unterteilt und eine effiziente Verteilung in heterogenen, asymmetrischen Clustern ermöglicht, wobei gleichzeitig eine einfachere Handhabung von Geräteausfällen und Performanz erreicht wird,

(ii) *ein heterogenes Cluster Computing Framework* - ***clusterCL*** - das unseren Partitionierungsansatz unterstützt, indem es eine Abstraktionsschicht zur Verfügung stellt, um die Details der Programmierung asymmetrischer, heterogener Cluster zu verbergen,

(iii) *ein Set optimaler Algorithmen für die Arbeitsaufteilung*, die Ausführungszeit und Energieverbrauch in einem Zeit-Energie-Lösungsraum berücksichtigen und den Ausführungsprozess Richtung Laufzeit, Energieeffizienz oder einem Kompromiss zwischen beiden leiten,

(iv) *Auswertung der Validität und Effizienz unseres Ansatzes* auf der Grundlage von Experimenten mit elf HPC-Anwendungen und zwei synthetischen Benchmarks aus verschiedenen wissenschaftlichen Bereichen und Benchmark-Suiten getestet auf drei mittleren, unterschiedlichen und asymmetrischen Clustersystemen mit heterogenen Computing Devices.

Acknowledgments

First and foremost I would like to thank my advisor prof. Eduard Mehofer. Without his continuous support, advice, and encouragement this thesis would have never been finished. In particular, I am grateful for his patience in discussing thoroughly the challenges of this thesis and for teaching me how to look at problems from multiple perspectives.

I am grateful to prof. Siegfried Benkner for his advise and support during my PhD studies, and to prof. Blerim Rexha for his personal and professional guidance throughout.

I am deeply indebted to my parents Jeton and Fatmire for their trust and for supporting my academic career, and my sister Vlora for her constant encouragement and support.

Last but not least I would like to thank my wife Agnesa for her endless patience, love and support, and my children Jeta and Diell for their unconditional love and for always understanding that I had to go to work on some weekends instead of playing with them.

The research leading to these results has been financially supported by OeAD HigherKos Program and Research Group Scientific Computing of the University of Vienna.

Bibliographic Note

The chapters of this thesis are based on the following publications and manuscripts:

- **Chapter 3 and 5**: Valon Raca and Eduard Mehofer. "clusterCL: Comprehensive Support for Multi Kernel Data-Parallel Applications in Heterogeneous Asymmetric Clusters", submitted, 2019.
- **Chapter 4**:Valon Raca and Eduard Mehofer. "Device-Sensitive Framework for Handling Heterogeneous Asymmetric Clusters Efficiently". In: International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD). 2015, pp. 178-185.
- **Chapter 6**: Valon Raca and Eduard Mehofer. "Efficiency considerations in heterogeneous cluster systems". In: 18. Kolloquium Programmiersprachen und Grundlagen der Programmierung (KPS). 2015, pp. 515-524.
- **Chapter 7**: Valon Raca, Eduard Mehofer, Marcus Hudec. "Optimal Time and Energy Efficient Work Distributions in Heterogeneous Systems". In: Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP). 2016, pp. 440-447.
- **Chapter 7 and 8**: Valon Raca, Eduard Mehofer, Seeun William Umboh, Bernhard Scholz. "Optimal Scheduling for Deploying Work Packages on Heterogeneous Distributed System", submitted, 2019.

Contents

List of Figures

List of Tables

Part I.

Foundations and Motivation

CHAPTER 1

Introduction

The necessity for more computational power has been recognized since the beginning of the computer era. With the increasing complexity of algorithms and applications which help to solve scientific problems and address industrial challenges, the efforts to increase computational power have been central to the invention in computing technology. This has driven chip manufacturers to produce CPUs which were faster and cheaper in order to meet the demand of the market. Until a few years ago, the mainstream in High Performance Computing / Supercomputing has been homogeneous clusters and supercomputers consisting of many nodes with powerful CPUs. Now, the mainstream has shifted toward Heterogeneous Computing where heterogeneous devices such as GPUs augment the role of the CPUs in solving composite problems which otherwise would take a long time to be processed by CPUs alone. The latest supercomputer machines contain heterogeneous devices along with the CPUs, with most of the top ten supercomputers as of late 2018, being based on heterogeneous hardware [Top18].

Heterogeneous devices have a very different architectural design from CPUs, e.g. GPUs have a massive number of lightweight cores, APUs combine a CPU and a GPU in a single die, Xeon Phi has many x86 compatible cores connected via a ring bus, FPGAs have arrays of logic blocks which differ significantly from the design of a CPU. These architectural differences prevent the execution of x86 software in these devices. Unlike complex designs of CPUs, heterogeneous devices are built as special-purpose processors with a clear strategy in mind to tackle specific problems. GPUs as the most prominent representatives of heterogeneous devices have been primarily designed to support a massive number of floating-point calculations per video-frame due to pressures from the fast-growing video-gaming industry. Thus, GPUs are equipped with hundreds of small, basic floating-point execution units designed to maximize throughput. Owing to the specific underlying architectures of those devices, some types of

programs such as data-parallel applications can be executed in these devices much faster [KH12]. Another advantage of using heterogeneous devices is that they are usually much more energy-efficient. Processing units of those devices have a much simpler design requiring less energy to be powered efficiently. These two advantages have made heterogeneous devices hardware of choice for building High Performance Computing systems.

However, there ain't no such thing as a free lunch. These systems are very hard to program. Heterogeneous devices have been in a different path of development in comparison to CPUs which are designed to optimally execute complex sequential codes [GHK+12, KH12]. Differences in hardware characteristics between CPUs and heterogeneous devices prompted the development of new programming platforms to support execution in heterogeneous platforms. Initially, a programming model supporting general-purpose programming in GPUs, CUDA [nVi19a], has been developed by nVidia. It is used to execute data-parallel applications exclusively in CUDA-enabled devices, being the GPUs produced by nVidia [KH12]. The urge to support other types of accelerators and GPUs led to an open standard, OpenCL [Khr14], managed and supported by a consortium of major industry manufacturers, the Khronos Group. OpenCL as an open-source standard for general-purpose parallel processing in various processors left behind the era of non-portability of applications in heterogeneous systems, enabling applications to be executed in all OpenCL-supported processors. However, performance portability remains an issue to this day. Developing efficient and portable applications requires hand-optimization of the code to compensate for differences in the architectural design of the targeted processors [OWG13].

In simpler heterogeneous platforms such as computers that host a CPU and a GPU, the computing process is handled by OpenCL where CPU issues commands to coordinate the execution process and move data, while the GPU executes the parallel code. Most of HPC programs programmed in OpenCL target single devices or to an extent single node cluster systems. In a cluster system, this leads to the idleness of the rest of the system, resulting in under-performance due to the inability to utilize all available computing resources. It has been shown in [GPCF14], that aggregated performance from a whole system decreases execution time in both homogeneous systems and heterogeneous systems. Therefore, aiming for execution of parallel applications by distributing work in all available cluster resources is very important. Nonetheless, supporting execution in a cluster system is very difficult with challenges such as the impact of data transfers, workload balance, and programming support. Inter-node communication is inevitable and being an extra layer of data transfers it may significantly affect the overall execution process, especially for data-intensive applications. Achieving workload balance is another challenge in heterogeneous systems due to different computing capacities of the devices and variable memory capacities and organizations. Next, many programming plat-

forms should be mixed into a hybrid programming approach to provide support for executing applications in a cluster. Such a hybrid programming approach with MPI and OpenCL complicates programming considerably, even in the context of symmetric nodes with identical hardware configurations. Typically, MPI codes assume symmetric nodes resulting in an even distribution of work onto the nodes making it easier to ensure load balance. However, nowadays most of the heterogeneous clusters in computational labs are asymmetric with different kind of accelerator devices requiring new strategies in dealing with such systems. In this thesis, we focus on handling heterogeneous asymmetric clusters while providing an abstraction layer for the programmer and enable seamless computing of HPC applications.

Due to the redundancy and the massive number of processing cores, especially in GPUs, data-parallel applications can harvest better the performance of the heterogeneous architectures. Many threads created by heterogeneous devices allow for much finer granularity of execution. However, except for embarrassingly parallel applications, different applications may have different computational behavior in different devices. Transforming these applications to execute in different devices of the cluster system requires work partitioning. Differently, from the homogeneous systems where workload partitioning is often trivial, many obstacles arise from partitioning applications in heterogeneous systems, mainly due to computational behavior of the applications and device characteristics. Hence, partitioning strategies have to be used carefully and must take into account the characteristics of the targeted application such as data access patterns. Distribution of work follows partitioning, with static and dynamic approaches used to guide the scheduling of work in devices and improve performance. However, considering only load balance and execution time is not sufficient anymore. In addition to runtime efficiency, heterogeneous devices raise an important issue – energy efficiency. Depending on the computational characteristics of an application, runtime and energy efficiency may differ considerably between different types of compute devices. Optimizing in the direction of the execution time or in the direction of energy consumption neglecting the other dimension is not enough. Efficiency considerations have to take both dimensions into account at the same time leading to new optimization problems. We consider in detail those optimization problems in this thesis and provide optimal solutions for work distribution.

1.1. Problem Statement

Our goal is to efficiently execute data-parallel applications in heterogeneous asymmetric clusters by optimizing for execution time and energy efficiency. Targeted architectures in this thesis are mainstream nowadays, while programming them is a very complex task. Applications that most benefit from heterogeneous architectures are very diverse with features that require specially

tailored strategies to improve performance. Finally, optimizing only for performance is not enough. Current systems consist of devices of variable energy-efficiency coefficients, hence optimizing in the direction of energy efficiency is necessary.

In this thesis, we address these issues guided by the following research questions which reflect the problems of programming and optimizations in those environments.

RQ1. **How to assign the right amount of data/work to heterogeneous devices?** OpenCL kernels are usually written to be executed in a single heterogeneous device. Hand optimizations of the parallel program targeting a specific device are usually sufficient for performance improvement. Harvesting performance from multiple heterogeneous devices requires a more sophisticated approach. Application data has to be partitioned between devices and considering the heterogeneity and diversity of the devices, partitioning and distribution of work is not a trivial problem. Hence, the approaches, strategies and methods for partitioning and distributing of work have to be cautiously analyzed and devised to target both the hardware architecture and application characteristics specifically. An important issue in this context is ensuring workload balance either between devices or between cluster nodes in order to ensure synchronized completion of execution in the whole system.

RQ2. **How to support partitioning and scheduling?** OpenCL and MPI as parallel programming platforms for heterogeneous platforms cannot by themselves solve all the problems. OpenCL enables porting of applications to devices and MPI supports data transfers between nodes. However, between them, there is a whole middle layer which does not exist. In the absence of this layer, the programmer is required to take care of many tasks such as handling different devices, partitioning and distribution of work, coordination of execution, workload balance, dealing with communication overhead, handling device failures and merging results to name a few. Bridging this gap would provide scientists and engineers with an abstraction layer towards the hardware architecture of the system and programming complexities of such systems. It would allow them to focus on translating domain-specific problems into data-parallel applications which further can be handled automatically by a framework.

RQ3. **How to deal with applications with different kind of dependencies?** The vast majority of data-parallel applications do not fit into the category of embarrassingly parallel programs. They are not easily decomposable into small independent tasks. Partitioning of data-domain into smaller chunks may result in inter-device communication since the code logic of the application may require accessing data which are not readily available in device memory. This would lead to extra communication

overhead in order to ensure computing correctness. Since heterogeneous devices do not act like a CPU and explicit commands have to be issued to check and monitor execution, the overhead in communication and coordination will increase, raising the programming complexity. Moreover, data-parallel applications usually consist of many data-parallel regions of code introducing data-dependency problems between these regions. All these problems are application-centric stemming from application characteristics. Therefore, it is vital to analyze different patterns of computational behavior of data-parallel applications in order to come up with more generic categories of applications for which a set of comprehensive strategies dealing with dependencies can be applied.

RQ4. How to optimize program execution taking both execution time and energy consumption into account? Heterogeneous devices besides being more time-efficient than CPUs in executing massively parallel programs, have been shown to be also more energy efficient. This comes mainly as a result of simpler designs of the processing elements of these devices, which are more energy-efficient. Design of the accelerators and especially of GPUs is driven by increasing performance per Watt, which makes energy consumption an important factor of the computing process in heterogeneous clusters. Both runtime efficiency and energy efficiency inevitably need to be considered when optimizing execution and devising work distribution strategies. Depending on which direction the optimization is steered, different scheduling strategies are possible. A good approach in this context would enable optimal mapping decisions taking into account user goals and constraints that can be calculated efficiently and if required dynamically while the program is executing.

RQ5. How to recommend reasonable trade-off solutions between time and energy? Workload partitioning process is constrained by the granularity of parallelism. For applications which exhibit high arithmetic intensity and large datasets even the heterogeneous devices struggle to perform. This tilts the partitioning decisions towards finer grain parallelism, leading to smaller chunks of work assigned to devices. With the number of chunks outnumbering the number of devices by orders of magnitude, optimizing work scheduling strategies becomes difficult. The set of scheduling solutions reflecting all possible mapping decisions becomes enormous and examining each of those solutions takes a lot of time. This is the most simple approach towards recommending a trade-off solution between performance and energy consumption for work scheduling. However, selecting a solution with an exhaustive search algorithm is impractical. A better approach would be to reduce the set of solutions to contain only feasible solutions and from there select a solution which would be the best trade-off between execution time and energy consumption.

1.2. Contributions

The work on this thesis has been focused in answering a set of problems introduced in the field of Heterogeneous Computing as outlined in Section 1.1. The main contribution of this thesis is a framework that bridges the current gaps to better utilize heterogeneous architectures with a special focus on a work partitioning approach that can ensure workload balance in an asymmetric cluster while enabling and recommending scheduling solutions which are optimized for runtime efficiency and energy consumption. More precisely, the contributions of this work are as follows:

C1. A generalized approach for workload partitioning in heterogeneous asymmetric clusters that ensures workload balance

C2. Design and implementation of a framework to support seamless computing in heterogeneous clusters

C3. Analysis of computational patterns of HPC applications and implementation of mechanisms to handle applications with dependencies

C4. Optimal scheduling algorithms to optimize for performance and energy-efficiency

The first major contribution is our approach in handling workload partitioning for data-parallel applications in heterogeneous systems. Our approach is based on a strategy to ensure workload balance between the cluster nodes instead of a customized approach which ensures synchronized completion at device level. Data chunks created by this partitioning process can oscillate in size between fine grain to coarse grain partitions taking into consideration the arithmetic intensity and the size of the dataset for a particular application.

To overcome the gaps between programming models for heterogeneous clusters, we have implemented a framework that provides an abstraction layer for the underlying architecture of the system. The framework is split into three major components which correspondingly control the partitioning and work distribution process, coordinate the execution in nodes, handle device failures, and steer the execution process in the devices. The programmer is only required to specify data and choose one of the partitioning strategies that fits better for the application, while the other part is handled automatically by the framework.

Beyond the challenges posed by hardware architecture of the heterogeneous systems and the programming support, application characteristics play a major role in influencing the execution process both in terms of performance and the practicality of programming. We have analyzed computational patterns of a broad range of HPC applications with a special focus on inter-device (intra-kernel) and inter-kernel dependencies. We have extended the framework to support computing for generic multi-kernel HPC applications by implementing

synchronization mechanisms to ensure data coherency while minimizing the impact of additional data transfers.

Finally, we have designed optimal algorithms that target work distribution in clusters with a higher degree of heterogeneity and asymmetrical configuration. The algorithms take into account two objectives for optimization of the computing process: performance and energy efficiency. The scheduling process can be steered in any of the directions, while it can also recommend a distribution strategy that is based on the trade-off of getting better performance while conserving energy. The algorithms have been implemented and support work distribution decisions of the framework.

Publications and submitted manuscripts that are based on the work presented in this thesis are as follows:

1. V. Raca and E. Mehofer, Device-Sensitive Framework for Handling Heterogeneous Asymmetric Clusters Efficiently, 26th IEEE International Symposium on Computer Architecture and High Performance Computing (Florianopolis, Brazil), Oct 2015, pp. 181–188.

2. V. Raca, E. Mehofer, Efficiency Considerations in Heterogeneous Cluster Systems, 18. Kolloquium Programmiersprachen und Grundlagen der Programmierung - KPS 2015, (Pörtschach am Wörthersee, Kärnten, Austria), Oct 2015.

3. V. Raca, E. Mehofer, and M. Hudec, Optimal Time and Energy Efficient Work Distributions in Heterogeneous Systems, 24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP) (Heraklion, Greece), Feb 2016, pp. 440–447.

4. V. Raca, E. Mehofer, Implementing HPC Applications with a Framework supporting Heterogeneous Asymmetric Clusters, [submitted 2019]

5. V. Raca, E. Mehofer, clusterCL: Comprehensive Support for Multi Kernel Data-Parallel Applications in Heterogeneous Asymmetric Clusters, [submitted 2019]

6. V. Raca, E. Mehofer, B. Scholz, W.U. Seeun, Optimal Scheduling for Deploying Work Packages on Heterogeneous Distributed System, [submitted 2019]

1.3. Outline of the Thesis

This thesis is organized in four parts, as shown in Figure 1.1.

Part I, respectively Chapter 2 gives an overview of the current hardware technologies and systems, programming models for heterogeneous clusters and describes data-parallel applications which are the object of this work.

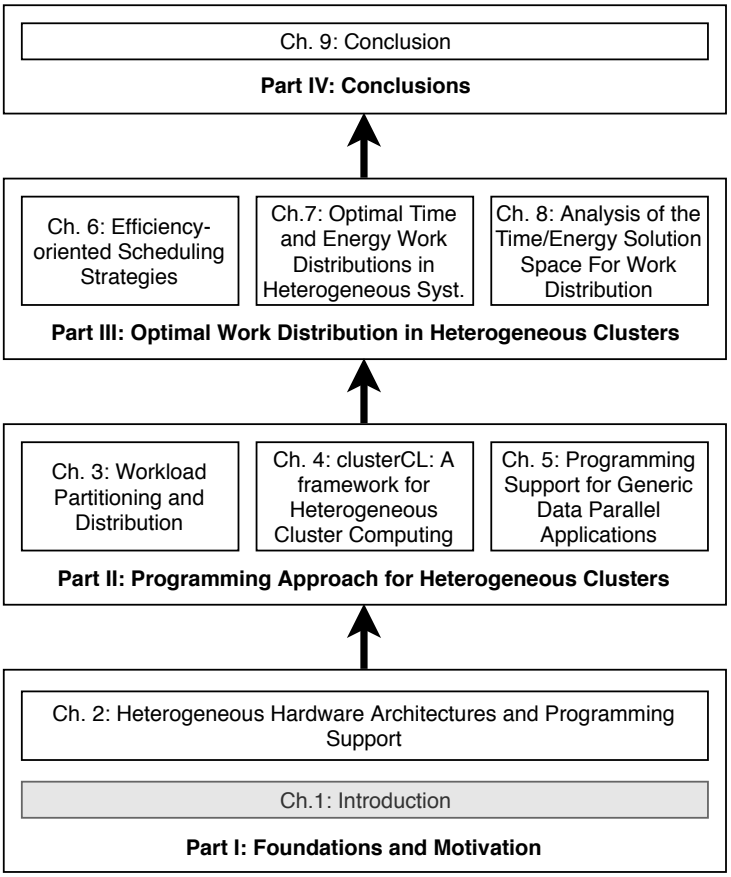


Figure 1.1.: Outline of the Thesis

Part II consists of three chapters where we present in detail and evaluate our programming approach for heterogeneous asymmetric clusters. In Chapter 3, we focus on the approaches for workload partitioning as means to ensure workload balance in a cluster system. Chapter 4 presents our clusterCL framework which provides an abstraction layer to the programmer for the underlying hardware architecture. In Chapter 5 we analyze some features of HPC applications and introduce strategies to handle generic applications with dependencies.

Part III is focused on work distribution and scheduling strategies which provides the programmer with a platform to schedule work, taking into account both execution time and energy efficiency. Chapter 6 sets the requirements and the optimization model for the clusterCL framework, while optimal algorithms satisfying different optimization criteria are presented in Chapter 7. In Chapter 8, we discuss different strategies for recommending solutions based on a trade-off between execution time and energy efficiency.

Finally, in Part IV we conclude with a summary of our findings and discuss the future directions of programming support for heterogeneous architectures.

CHAPTER 2

Heterogeneous Hardware Architectures and Programming Support

In this chapter we give an overview of the state of the art heterogeneous hardware architectures and the programming support that enables the execution of applications on these platforms. We also tackle separately in this chapter the application domain, particularly data-parallel applications which has benefited most from the use of heterogeneous devices.

2.1. Heterogeneous Hardware Architectures

Heterogeneous hardware architectures nowadays are present in almost all computing systems, starting from small embedded systems to the biggest supercomputers of the world. These systems include special-purpose processors which differ in many aspects from the design and the architecture of the traditional CPUs. Although the goal has never been to replace the role of the CPUs in computers and other computing systems, heterogeneous devices are getting an ever-increasing share while complementing CPUs in the computing process, especially for applications or parts of applications for which those devices have special circuitry and are specialized to process data faster. Further, using a combination of many of those devices for running applications have been shown to increase performance despite increased communication mainly due to data transfers in new routes such as inter-node communication in a cluster system.

In the next sections, we introduce the hardware architecture which we target in our work. We discuss heterogeneous technologies and the cluster systems consisting of multiple heterogeneous devices and asymmetric configurations.

2.1.1. Heterogeneous Devices

Since the single-core era has reached its physical limitations, new approaches mainly multi-core and many-core have been explored [CDDE13]. Exploiting the capabilities of various processors as accelerators has resulted in reaching high performance and at a much smaller price (at least in terms of energy consumption). This has benefited many fields of science and engineering, mainly contributing to high performance matrix-oriented computations in scientific computing, but also providing for multimedia processing, embedded systems or real-time systems [CDDE13, GHK+12].

Heterogeneous devices have a completely different underlying architecture and have been in a different path of development in comparison to CPUs which are designed to execute complex sequential code optimally. In Figure 2.1 we have shown a high-level architectural organization of a CPU, GPU, and an FPGA. The CPU in the figure consists of four powerful cores with a shared cache, DRAM, and control unit. Its cores have a complex design which serves to compute programs that require execution of the code containing complex logic. GPUs, on the other hand, consist of multiple streaming multiprocessors which in turn contain many smaller cores. These cores are basically arithmetical-logical units (ALU) that can process simple computations. All cores of a streaming multiprocessor usually share a control unit and cache. Access to the graphic DRAM (GDRAM) is possible from all compute units. FPGAs consist of many programmable logic blocks which can be reconfigured in such a way to perform combinatorial functions or act as simple logic gates.

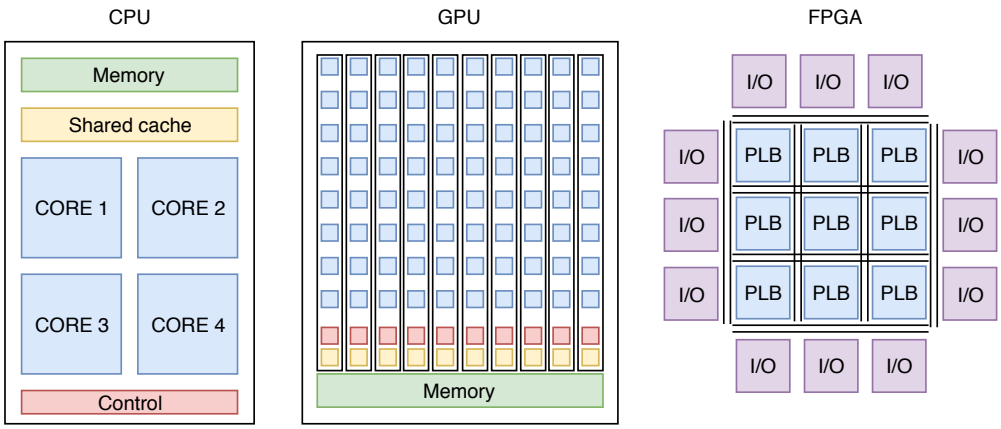


Figure 2.1.: Hardware architecture of a CPU, GPU and a FPGA

The most striking difference between a CPU and GPU from the organizational point of view is the difference in the amount of ALUs. GPUs can schedule a large number of threads which execute on those small cores. The number of cores in a GPU is usually in the hundreds with the number of threads that can be scheduled in the thousands. This makes GPUs a preferred device for

executing codes which contain a massive amount of simple parallel instructions. Lets consider Figure 2.2. In this figure, we see two types of applications *App 1* and *App 2* — squares present code blocks. Multiple squares in a horizontal row depict a highly parallelizable code region. Let us assume for the sake of simplicity that those regions are iterative loops of simple instructions. From the figure we understand that App 1 has more parallelizable code regions than *App2*. Further, those regions in *App 1* have a greater amount of repetitions than in *App 2*. Hence, *App 1* is more suited to be executed in a GPU through loop fission and transformation than in a CPU while the opposite can be observed for *App 2*. Moreover, a mixed approach is possible where sequential regions are executed in the CPU, while parallel regions are transformed and executed in a GPU.

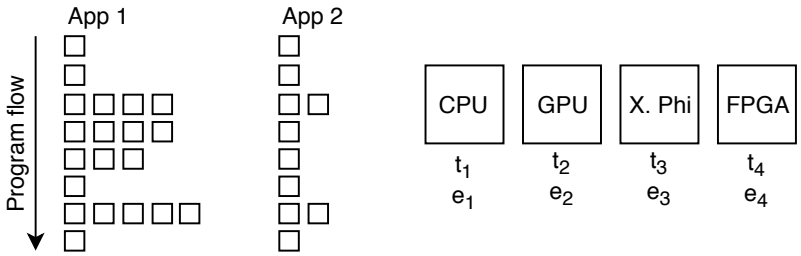


Figure 2.2.: Variability in execution time and energy consumption for CPU and different types of devices

On the right side of the figure 2.2 we see four different devices with respective execution times t_1, t_2, t_3, t_4 and energy consumption e_1, e_2, e_3, e_4 for a specific application. If we simplify the case to the most basic settings, we would in general have *App 1* execute faster in a heterogeneous device due to its parallel regions of code, while *App 2* being an application with mostly sequential code would execute faster in a CPU. As per energy consumption, this is less clear even in the most basic settings since the energy consumption is not usually correlated with performance and is based on the actual utilization of different units of the CPU or GPU. For this example, we used the most basic settings while there are many hardware, software and application-specific constraints which affect the computing process. Heterogeneous devices besides consisting of more processing units do not share many other features with the CPUs such as they have different instruction set architectures, memory organizations, communication interfaces, programming approaches, and so on. Therefore, execution time and energy consumption are influenced by many factors among which also the application characteristics. Most importantly, many of these factors impact the two dimensions differently.

2.1.2. Heterogeneous Asymmetric Clusters

Large clusters consisting of heterogeneous computing nodes with different computing devices provide the opportunity to scale-up the performance. The hardware model we target in our work is shown in Fig. 2.3. The system consists of asymmetric cluster nodes with distributed memory, which are connected with Infiniband/Ethernet with a front-end node. *Cluster node 1* and *cluster node 2* are configured with five devices. *Cluster node 1* hosts three GPUs of different types (e.g. an nVidia Tesla K20m, an nVidia Tesla V100 and an AMD Radeon Instinct MI25) and two digital signal processors (DSP) of the same type. *Cluster node 2* has a similar configuration hosting four GPUs, two of each type and an accelerator of type 2 (e.g. Intel Xeon Phi). *Cluster node 3* hosts four identical FPGAs, while *cluster node 4* consists of two accelerators of different types. It is clear from different amounts of devices in each node and from the diversity of the types used in the nodes that this cluster structure is asymmetric, much more different from the monolithic structures of the past homogeneous systems. However, these are not the only reasons for asymmetry. In *cluster node 1* GPU Type 2 is not supported by some programming platform used by the programmer while Accelerator Type 2 in *cluster node 4* is broken and has not been replaced yet resulting in a temporary asymmetry. Such situations occur rather often in practice preventing an even distribution of work over all cluster nodes.

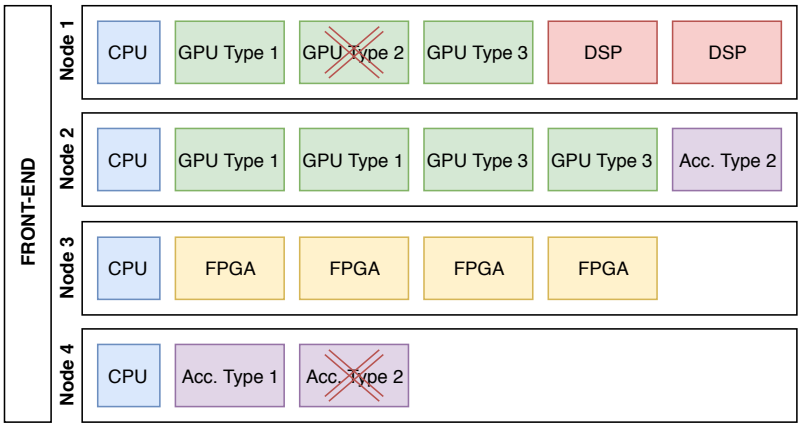


Figure 2.3.: Hardware architecture of a cluster with asymmetric nodes

Besides the heterogeneity and the asymmetric structure of a cluster, there are other issues which are important when dealing with cluster systems. One of the most important is the impact of data transfers in the overall execution process. Data transfer in a distribute-memory architecture as presented in Figure 2.4 occur in two levels: from the CPU DRAM to the device memory and from front-end to the cluster nodes including here also the inter-node communications. Typically the heterogeneous devices are discrete cards with separate

DRAM connected via PCIe to the CPU. This type of connection requires explicit data transfers. Throughput in this channel of communication is much lower than between CPU and DRAM, respectively between processing units and device memory in heterogeneous devices. Additionally, latency is much higher in PCIe channel. However, data transmission rate between the cluster nodes via InfiniBand or Fast Ethernet is the slowest in the whole system corresponding only to a fraction of the throughput of the internal transfers within a device or PCIe transfer rate. Differences in transmission rates can be a decisive factor in performance, especially when executing some types of parallel applications which require a lot of data while having low arithmetic intensity.

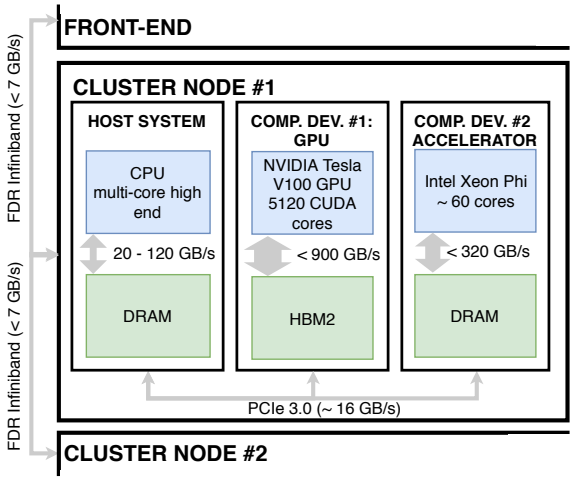


Figure 2.4.: Data transfer channels in a cluster system

2.2. Software Architecture

A cluster system as described in the previous section requires multi-paradigm programming models to support application execution. Since none of the standards (MPI, OpenMP, OpenCL, CUDA etc.) offer a platform which would enable heterogeneous computing in heterogeneous architectures, a combination of several platforms is required to handle these systems. From a bottom-up approach, a programming model to support execution of the applications in heterogeneous devices is required. Currently, there exist a few programming models that enable low-level heterogeneous computing with the most well known being CUDA and OpenCL. Many studies [KSA+10, FVS11, SCL+12] have looked into performance comparison between CUDA and OpenCL technologies for different heterogeneous devices and different applications. The results show small differences in execution time between the two. The advantage of using OpenCL over CUDA to execute parallel applications lies with the

OpenCL's ability to work with different hardware (CPU, GPU, Accelerators) from different vendors while CUDA is a proprietary programming model of nVidia which works only with nVidia GPUs.

Some studies [GTO13, KHD+17] have used mixed platforms for programming CPUs and heterogeneous devices, with the multithreading OpenMP [Ope19b] used for CPU execution to harvest the performance of multi-core processors and OpenCL used to program GPUs and accelerators. [SFSV12, SCL+12, SPB16, LPB18] have compared and discussed the performance of OpenCL and OpenMP and the results show that OpenCL and OpenMP are most of the time on par for CPUs, while other studies show that either OpenCL or OpenMP are slightly better. However, multithreading libraries as OpenMP or pThreads may be used for another purpose. With many devices on the same node, a multithreading library is required to handle device requests promptly, such that no device waits for the other to complete explicit data transfer or program execution.

In a cluster system communication between cluster nodes is necessary. The most widely used standard for communication is the Message Passing Interface (MPI) [MPI12]. It has been shown that MPI, although error-prone, can provide good performance in communication between cluster nodes and is highly scalable [HSdSM10].

In our work, we use a combined software architecture of MPI, OpenMP and OpenCL to target heterogeneous clusters. In the next sections, we give a short overview of these programming models.

2.2.1. OpenCL

Differences in hardware architecture between CPUs and GPUs required a new approach towards programming massively parallel processors. The necessity to support a broader range of GPUs and accelerators produced from different vendors led to an open standard, OpenCL, managed and supported by a consortium of major industry manufacturers, the Khronos Group. OpenCL provides an API which is general enough to support execution in different architectures.

Along with the API, the OpenCL framework provides a language, libraries and a runtime support for software development. OpenCL runtime support is responsible for host-device communication and generates device-specific binaries out of the OpenCL kernels, while its programming language, a restricted version of C99 language with extensions to support data-parallel codes [GHK+12], offers a hardware abstraction layer, enabling the development of kernels without prior knowledge of the underlying architecture of a computing device.

The OpenCL platform model specifies a host-accelerator model where the execution has to be coordinated by a host (CPU), while the data-parallel code (OpenCL kernel) is executed on one or more devices using vendor-specific implementations of the OpenCL API. As shown in Figure 2.5, each accelerator is modeled as a Compute Device. Compute devices are divided into Compute

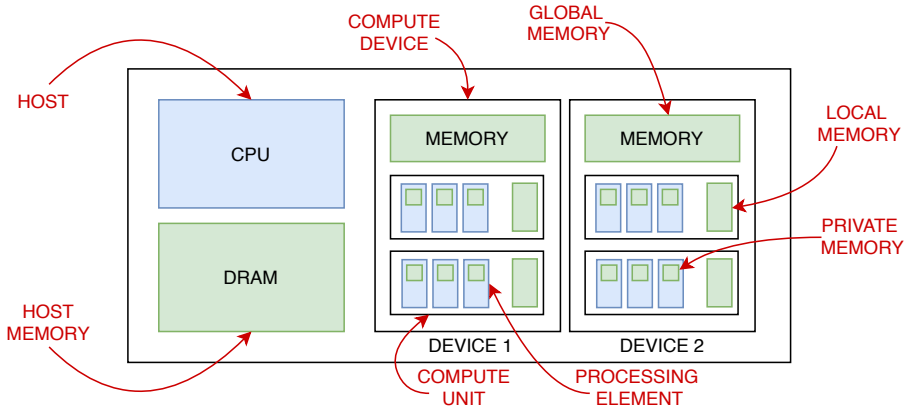


Figure 2.5.: OpenCL Terminology

Units (e.g. streaming multiprocessors in GPU) and further divided into Processing Elements (e.g. GPU cores).

OpenCL Memory Model resembles a data-parallel execution model. Besides host memory (DRAM), a memory hierarchy is defined for compute devices with global memory shared by all processing elements, constant memory which is at the same level as global memory but for read-only access. Each compute unit has its own local memory which can be shared among work-group threads, while processing elements have their own private memory which is accessible only by the work-item executing in the processing element.

The execution in OpenCL supported processors is coordinated by the host program, which enqueues to the targeted computing device an N-dimensional index-space of kernel instances (NDRange) mapped to the processing elements of a computing device in the form of threads of execution (work-items). These work-items are grouped into work-groups which are assigned to compute units [GHK+12, KH12, TKS+11, MGM+11].

Below is given a simple example adding two arrays in C and OpenCL. In C a loop is required to go over array elements. In OpenCL, each thread executes a single kernel instance in this case adding an element from each array. Parallelism granularity can be adjusted according to the requirements.

Listing 2.1: C version

```
1 void addArray(int *a, int *b, int *c, size_t n)
2 {
3 for(int i = 0; i < n; i++)
4 {
5 c[i] = a[i] + b[i];
6 }
7 }
```

Listing 2.2: OpenCL version

```
__kernel void addArray(__global int *a,
__global int *b, __global int *c, size_t n)
{
int tID = get_global_id(0);
c[tID] = a[tID] + b[tID];
}
```

2.2.2. Multithreading

Multithreading libraries provide support in creating threads within the context of one process. Many libraries support multithreading with the POSIX threads or pThreads [fS09] providing a standard library for multithreading. Higher-level languages such a Java, C++, Python enable developers to use multithreading. Other programming languages are built around the concept of concurrent programming and multithreading and expedite programming with threads such as OpenMP, CILK [Int17] and TBB [Int19]. However, in our context, multithreading is used to concurrently manage device within a node such that explicit OpenCL calls to instruct device drivers for an operation are handled concurrently by the process. For this purpose, any programming language or language extensions and libraries that support concurrent programming can be used.

2.2.3. MPI

When it comes to communication between cluster nodes or simply communication between processes in distributed-memory architectures, MPI is the most widely used parallel programming protocol. In Figure 2.6, the process of communication between two cluster nodes is shown. Data are sent from one process to the other via the network with the mediation of the system buffers. MPI calls typically are point-to-point communications involving two processes, one being the sending and the other receiving [Lib19].

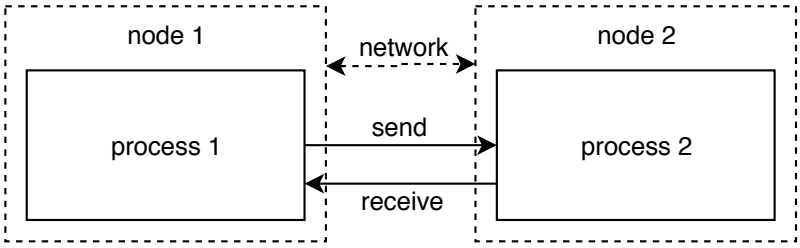


Figure 2.6.: MPI Communications

MPI also supports multithreaded MPI calls. In Figure 2.7 we have shown

the differences between single-threaded MPI calls where only the master thread in the sending process is allowed to issue MPI calls, and on the bottom a multithreaded approach where each thread in the sending process is allowed to issue MPI calls. Testing results of [TG07] show that for some implementations of MPI such as MVAPICH2 [Uni19] and OpenMPI [Con19] using the multithreaded approach has resulted in performance improvement.

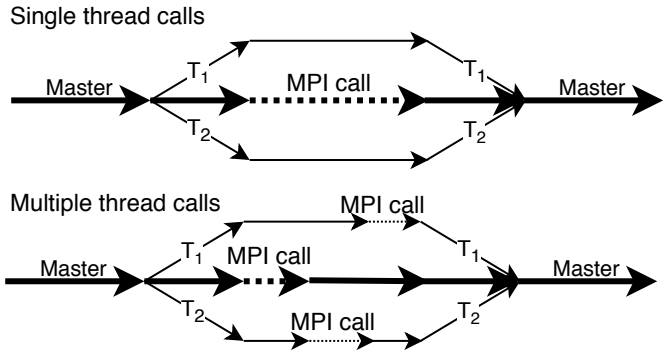


Figure 2.7.: MPI threading support

2.3. Application Domain

Applications that work best with heterogeneous devices, particularly GPUs are data-parallel applications. Data parallel applications are mostly found in scientific fields such as physics, chemistry and biology among others, but there are also applied engineering fields which have applications that are categorized as data-parallel application.

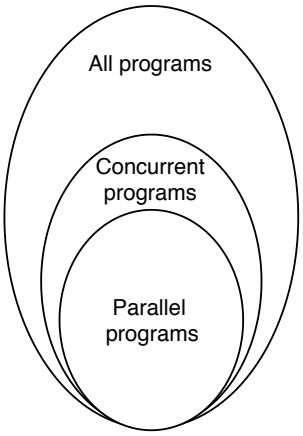
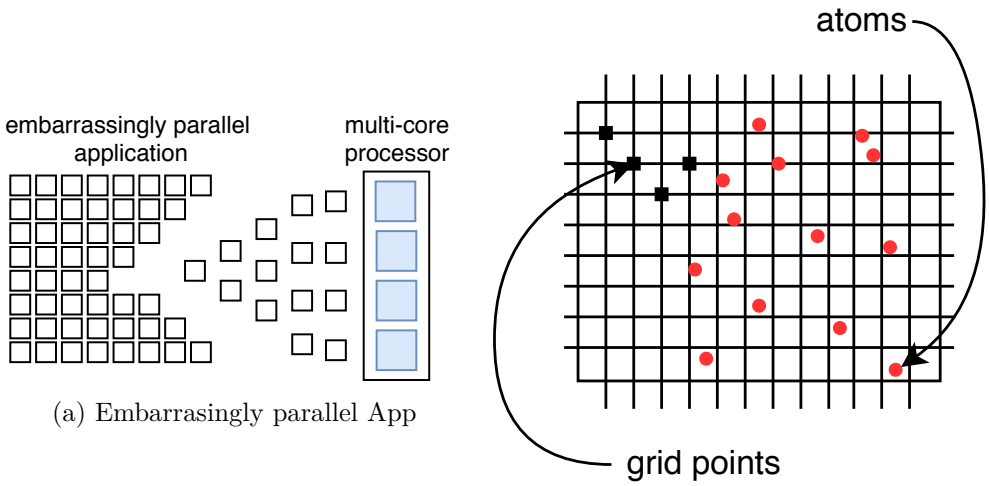


Figure 2.8.: Parallel programs as subset of concurrent programs and all programs [GHK+12]



(a) Embarrasingly parallel App

(b) Direct Coulomb Summation

Figure 2.9.: Data parallel applications

Data parallel applications usually have a large dataset upon which same or similar operations are performed. This is a perfect setting for GPUs which consist of hundreds of cores. It can provide thousands of threads that execute in parallel fashion while the same code logic applies to all running threads. Data parallel applications according to Flynn Taxonomy [Fly72] on parallel computer architectures are categorized as Single Instruction Multiple Data, or in a more general way Single Program Multiple Data (SPMD). Parallel programs are a subset of concurrent programs, since all parallel programs share the same properties of concurrent programs, although as shown in Figure 2.8 the opposite does not apply [GHK+12].

In Figure 2.9a, we have shown the execution process of an embarrassingly parallel application. These applications are the most basic data-parallel applications since the decomposition of the application into smaller parallel task is pretty straightforward and almost effortless. These mini-tasks are than mapped easily to processing elements of an accelerator.

However, most of the data-parallel applications are not embarrassingly parallel, and they usually consist of code logic that leads to some form of dependencies or communication between the decomposed units of the application. A simple case is Direct Coulomb Summation depicted in Figure 2.9b where the electrostatic potential is measured in a grid that encompasses atomic charges. Each of the black squares in the figure is a grid point where the influence of electrostatic potential is measured. Red circles show atoms which have their positions and charges. From a programmer point of view, each grid point can be assigned to a thread which would iterate over all atoms by calculating distances and influences from a particular charge. As long as all atom data and

grid data can be put in the device memory, this application is embarrassingly parallel. However, the number of atoms which are used for calculation can be very big and a good strategy to follow would be to partition the data and work between many devices. This strategy would increase performance since the workload can be distributed between devices. Nevertheless, the workload partitioning and distribution can introduce dependency issues no matter which strategy we use to partition data.

We deal in depth with partitioning strategies and work distribution for data-parallel applications in the rest of this thesis and our work is focused in providing solutions for partitioning and distribution of work in heterogeneous architectures.

Part II.

Programming Approach For Heterogeneous Clusters

CHAPTER 3

Workload Partitioning and Distribution

In this chapter, we discuss the approaches, strategies and methods for partitioning data-parallel applications and adapting workload for heterogeneous clusters. As a means of achieving better performance, we discuss the general work distribution approach that augments our work partitioning approach. We discuss in detail our work on work scheduling strategies and optimizations taking into account performance and energy efficiency in Part III.

Exploiting heterogeneous clusters requires applications to be partitioned, so the burden of computing is distributed among cluster's heterogeneous devices. This does not just increase resource utilization in the cluster but increases the performance since smaller workloads can be computed much faster and merged into the final output. However, contrasting from almost always trivial approaches of workload partitioning in homogeneous systems, the difficulty of workload partitioning in heterogeneous systems increases enormously with many factors to be taken into account such as the architectural designs of the devices, diversity of the devices in the system and configuration of the cluster among others.

Unlike for single device computations where the whole data domain is transferred to the device, partitioning and distribution of work in a multi-device asymmetric cluster system require addressing many challenges that arise in such settings. The problem domain has to be decomposed and depending on application characteristics, costly data transfers between devices and cluster nodes may be induced. However, the aggregated performance of all compute devices in a system usually is superior to the performance of the fastest device in the system despite communication overhead. An exception are heavily data-intensive applications.

Work distribution follows and augments the workload partitioning process

in improving overall execution performance. Work distribution strategies are crucial for avoiding the delay and unsynchronized completion of execution in cluster nodes and devices as well as in mitigating the effects of data transfer in computing.

Contributions

We present a workload partitioning approach for heterogeneous systems that partitions the workload into large number of same-size medium-grain chunks that can be mapped to heterogeneous clusters with different distribution strategies. Main advantages of our approach are as follows:

(a) *workload balance* between compute nodes can be ensured with distribution strategies

(b) *device failures and poor performance of the devices* can be handled quickly via rerouting chunks which can fit in all devices of the system

(c) *new optimizations strategies for performance and energy efficiency* can be implemented through this approach to steer the execution towards better performance and better energy-efficiency

Related Work

The related work is grouped into two categories: workload partitioning and work distribution.

Workload partitioning. Several works have proposed different methods for work partitioning in heterogeneous systems [LHK09], [GO11], [LSPM13], [GDF14], [SVL+16]. In [LHK09], Luk et. al. propose a method of adaptive mapping of work to the devices to find the best work distribution. Grewe et. al. [GO11] propose a machine learning approach to determine workload distribution in the presence of GPU contention, while Kofler et. al. [GDF14] use an offline training approach based on static and dynamic features of programs to decide on work distribution. Lee et. al. [LSPM13] propose SKMD framework which enables work partitioning for single node setting between CPUs and GPUs based on a heuristic method taking into account data transfer costs and execution time. Shen et. al [SVL+16] present an approach based on modeling, profiling and prediction techniques to determine optimal workload partitioning. All of these studies are based on parallel applications which are either single kernel applications or multi-kernel applications that have no inter kernel dependencies which would result in communication between devices or synchronization with the host.

Work distribution. Besides workload partitioning for HPC applications, distribution of work has been studied in several papers [Aug11], [PBAL13], [WWO14] and [LSM15]. These studies are mainly focused on dynamic task

scheduling. StarPU [Aug11] is a runtime system providing task-based scheduling for heterogeneous systems. Similarly, OmpSs [PBAL13] is a framework that requires annotating serial applications with directives in order to provide asynchronous execution of tasks in heterogeneous architectures. In [WWO14] Wen et. al. propose a task scheduling scheme which can schedule multiple kernels in different devices. In a similar fashion Lee et. al. [LSM15] propose a system for mapping different kernels to different devices while data are migrated based on the order of execution. These studies differ from ours since they are based on task-aware (kernel-level) partitioning and scheduling of applications to different devices while our approach is based on data-aware work distribution where the kernels are executed in serial fashion and inter-kernel dependencies are maintained.

3.1. Workload Partitioning

Scientific communities and engineering industries are perpetually hungry for performance. This demand cannot usually be fulfilled only by exploiting the performance of new, more powerful devices. An approach relying on a combination of processors and devices is required to achieve high performance. Partitioning of work among all CPUs in homogeneous systems has been at the center of High Performance Computing since the beginning. Large cluster and supercomputers hosting thousands of CPUs are used to speed up the execution process by partitioning applications into chunks and assigning them to CPUs while aggregating the results received from CPUs. Many benchmarks that are used to test the performance of cluster systems and supercomputers follow the principle of work partitioning to utilize all system CPUs. The same principle applies to heterogeneous systems.

To illustrate the performance increase from workload partitioning, we present in Figure 3.1 some measurements in our cluster CORA. CORA consists of seven identical nodes, each hosting a pair of Tesla C2050 GPUs and one Tesla C1060 GPU and nodes are interconnected with Fast Ethernet. The experimental settings are based in single device execution for Tesla C2050 and Tesla C1060 and are compared to the aggregated performance from workload partitioning in different configurations of the cluster. We use matrix multiplication routine (GEMM) to test for performance in different settings. We observe that Tesla C2050 is more than twice faster than Tesla C1060 for single device execution of GEMM (57.34s vs. 125.7s). If we partition GEMM, so the workload is distributed between devices, we observe a significant decrease in the overall execution time. A single node execution consisting of two Tesla C2050 and one Tesla C1060 (cf. Figure 3.1, CORA x1 NODE) with work distributed among them is 2.41× faster compared to a single device execution of the whole workload in the fastest device of the system (23.79s vs. 57.34s). Further, configurations with two and three nodes (cf. Figure 3.1, CORA x2 NODES and CORA

x3 NODES) reduce further the overall execution time being 1.82× (13.04s vs. 23.79s), respectively 2.35× (10.1s vs. 23.79s) faster than the execution in a single node setup. The speedup is smaller in the three-node version mainly because of the relatively short time of execution, considering that the setup and communication influence the execution process.

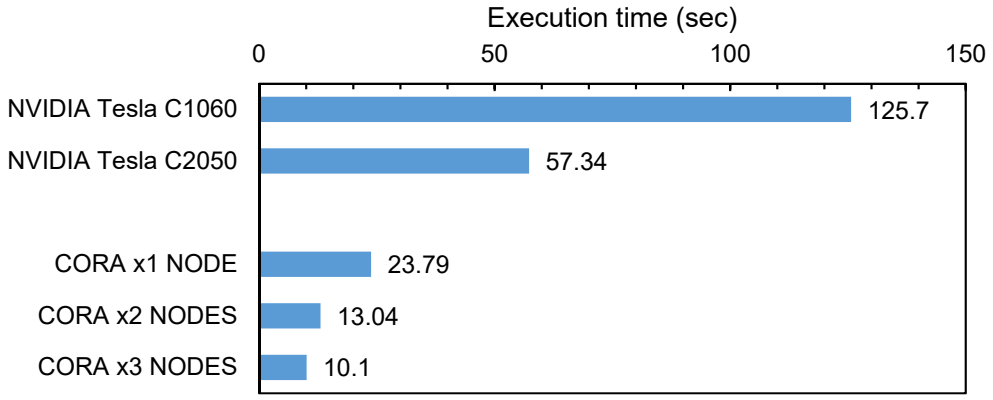
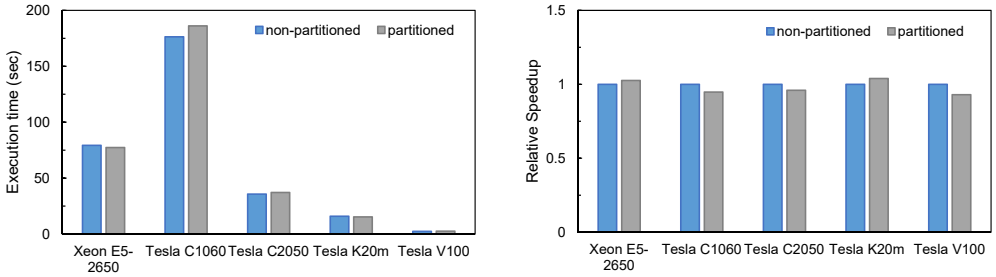


Figure 3.1.: GEMM: Single Device Execution vs Workload Partitioning in CORA cluster

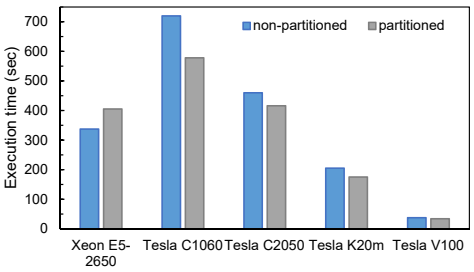
Additionally, we look at the potential overhead from the partitioning process itself. In Figure 3.2 we present the execution time and relative speedup for Direct Coulomb Summation (DCS) and Correlation Matrix (COR) in an experiment where first we execute the whole problem size with one device (non-partitioned execution), and then partition it into same-size data chunks and queue the partitioned chunks in the same device (partitioned execution). Therefore, we do not expect speedups; we just want to measure the overhead of the partitioning process. A small overhead from the partitioning process itself as well as from potential communication latencies is to be expected. For both applications, we use non-optimized kernels. As a baseline for speedup, we take the non-partitioned execution and compare with the partitioned execution.

For DCS we observe that the partitioned execution introduces a small overhead as expected such as for executions in devices Tesla C2050, C1050 and Tesla V100 (see Figures 3.2a and 3.2b). In some cases such as with the CPU and Tesla K20m it appears like the partitioning process is not causing any overhead and counter-intuitively speeding up the execution. In this case, our investigation shows that the speedup comes as a result of differences in process initialization and setup, hence no real speedup from the execution process itself. On the other hand, for COR we observe the opposite as shown in Figures 3.2c and 3.2d. In all devices, with the exception of the CPU the partitioned execution is much faster than the non-partitioned execution and the speedup is not negligible. Moreover, the speedup does not come as a result of differences in setup time but from the execution process itself. Although, it looks like an

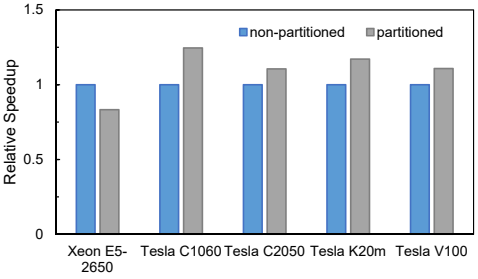


(a) DCS: Execution Time

(b) DCS: Relative Speedup



(c) COR: Execution Time



(d) COR: Relative Speedup

Figure 3.2.: Non-partitioned vs. Partitioned Execution in the same device

anomaly and an unexpected outcome, the reason behind the speedup is the computational behavior of the application. COR is an imbalanced application where the workload is concentrated in the upper triangular matrix which represents its dataset. Each work-item processes a row of the matrix of different length, causing the application to have an imbalanced spread of workload among threads. In the partitioned version we have partitioned the triangular matrix over the columns. As a result, this partitioning strategy has effectively produced a more balanced workload distribution between work-items. We will encounter in the later chapters similar problems not restricted to the domain of imbalanced data-parallel applications, where the selected partitioning strategy can impact the execution process significantly.

In general, the partitioning process itself and the cost of communications for data transfers between cluster nodes introduce an overhead in the execution process which, as seen from the previous examples inhibits us from achieving ideal speedups. However, the decrease in overall execution time from using many devices instead of only one is significant making partitioning a beneficial strategy for accelerating data-parallel applications despite the intricacies of the partitioning process and difficulties in programming heterogeneous clusters.

3.1.1. Partitioning Approach

Workload partitioning in homogeneous systems is simpler than in heterogeneous systems. Having all of compute devices (CPUs) with the same characteristics, workload partitioning is reduced to partitioning the application in same-size chunks which are assigned to all CPUs. This automatically ensures workload balance among CPUs of a compute node and further among compute nodes themselves. Partitioning is more difficult for heterogeneous systems. In Figure 3.3 we show the speedup of execution for different heterogeneous devices and a CPU compared to a baseline, which is the slowest device in the system. All devices are executing the same application with the same problem size.

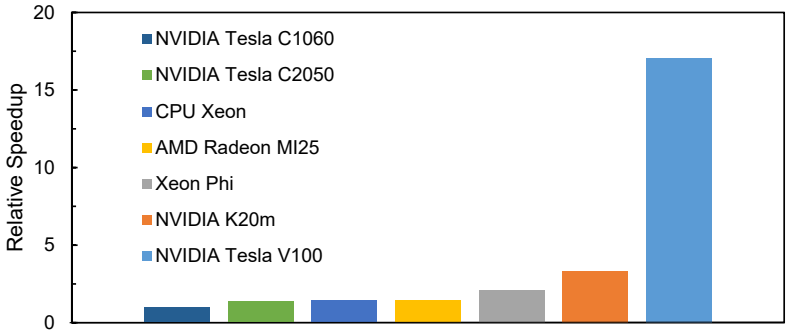


Figure 3.3.: Single Device Execution for Correlation Matrix in different heterogeneous devices and CPU

From the figure, we see the huge disparity in execution time between devices. The heterogeneity and diversity between devices can influence the workload partitioning greatly with significant consequences in the overall execution process. Many factors contribute to this. First, the devices differ in performance, memory organization, micro-architecture and other characteristics making the workload balance between the devices very challenging. Second, we target asymmetric clusters, consisting of different amounts and types of devices, making it difficult to achieve workload balance between the cluster nodes. Finally, the application's computational behavior influences both the partitioning strategies and execution time. For complex HPC parallel applications, partitioning induces communication between devices and nodes, which generates a lot of overhead in correctly and precisely mapping data accesses in a distributed memory architecture. Hence, a partitioning approach that addresses these issues is necessary.

In the domain of data-parallel applications workload partitioning and data partitioning usually point to the same thing. The general definition for data-parallel applications assumes that the same instructions apply to each data point of the application's dataset. Therefore, in most cases, workload partitioning reflects data domain partitioning. An exception are imbalanced applications

[SVL+16], which due to application characteristics have different workloads per data point.

Most of the studies for single-node settings prefer custom-size partitioning of data domain [LHK09], [LSPM13], [SVL+16]. In these papers different strategies such as profiling techniques and machine learning are used to adapt the data chunks to device capabilities better, although for a very small number of devices, usually a CPU and a GPU. In [BBD+19], Beaumont et. al. propose a method to support workload partitioning for a smaller number of heterogeneous devices, however in another paper [BBRR02] they show that the problem of workload partitioning for heterogeneous platforms with many devices is NP-Complete.

In general, data-parallel applications for heterogeneous systems can be partitioned using a *custom-size data chunks* approach, or *same-size data chunks*. The first approach for partitioning data-parallel applications is based on the assumption that the amount of work assigned to a device has to be tuned or predicted to reflect the computing power of the device. This can be approximated based on device's hardware characteristics, profiling information, or tuning. However, considering that the computing process in the devices is also influenced by the application structure, tuning workload partitioning to ensure workload balance can become quite complex. The second approach is similar to the strategy used in homogeneous systems where the work is partitioned in same-size data chunks. Since the partitioning process is the first step towards execution of data-parallel applications in a multi-device cluster it influences heavily the upcoming processes, especially distribution and as a consequence workload balance. We consider that in a cluster setting with a large number of devices, workload balance between cluster nodes should take precedence over the synchronized completion between devices. Therefore, a partitioning approach which shifts part of the problem of workload balance from the partitioning to distribution decisions might be useful.

We base our work on our partitioning approach, which breaks down data-parallel applications, respectively their datasets into medium-grained same-size-fits-all data chunks. The data chunks produced by this approach are almost never coarse-grained as in the custom-size partitioning approach but fluctuate between fine-grain and coarse-grain based on the arithmetic intensity on one hand and data intensity of particular applications on the other hand. Contrasting from the coarse-grain custom-size partitioning, our approach partitions the application such that the number of data chunks created by this process is always larger and in most cases orders of magnitude larger than the number of the devices in the system. Considering the big amount of data chunks to be distributed in a heterogeneous asymmetric cluster, workload balance becomes a problem of distribution strategies which have to ensure that faster devices get more data chunks than slower ones, balancing the work between devices and further between compute nodes of the cluster.

We have illustrated the major differences between the two approaches in Fig-

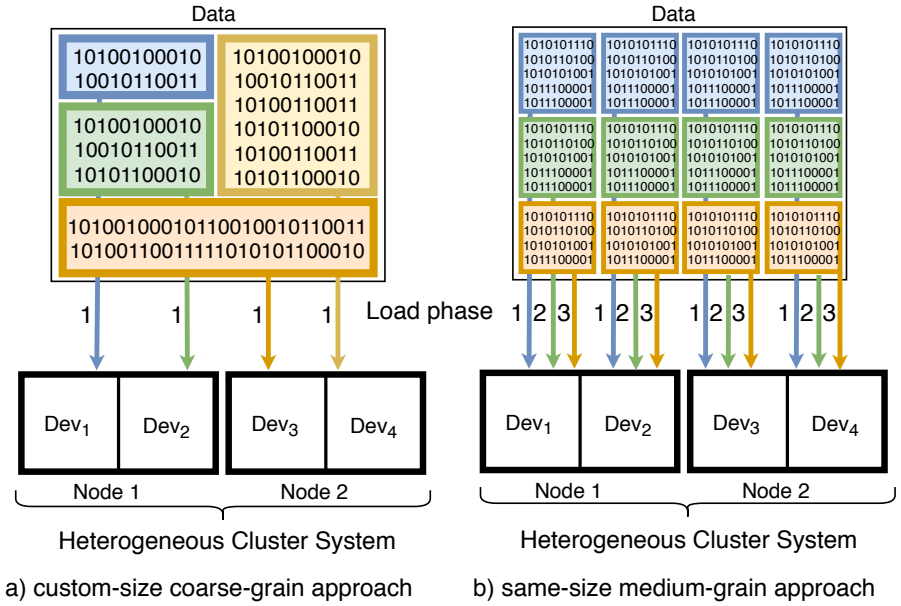


Figure 3.4.: Partitioning approaches: custom-size coarse-grain approach and same-size medium-grain

ure 3.4. On the left side, we see the custom-size approach where the partitioning has been tuned or predicted based on the hardware characteristics of the system's devices. We see that the largest portions of work have been assigned to devices Dev_3 and Dev_4, while the smallest ones to Dev_1 and Dev_2. There are only four data chunks each for one device which are launched at the same time. On the right side of the figure, we see the same-size medium-grain partitioning approach. The partitioning has resulted in many data chunks which are launched in different load phases to the devices, ensuring that the workload balance is achieved through the distribution of work instead of partitioning tuning.

Our workload partitioning approach produces same-size medium-grain data chunks, which are encapsulated into *work packages (WP)*. We use this term to refer to this type of data chunks through the rest of this thesis. Work packages are composed of a header and payload resembling the TCP/IP networking packets. The payload may retain the n-dimensional structure of data akin to the original data domain.

Shown in Figure 3.5 is a work package created by the workload partitioning process. The header includes information and instructions for the work package, which is essential for routing and failure handling. Each work package is given an ID which is used for logging and tracing by the runtime. Partitioning information stores data about the number of original dimensions, size of the payload and offset of the payload which indicates the displacement from the ori-

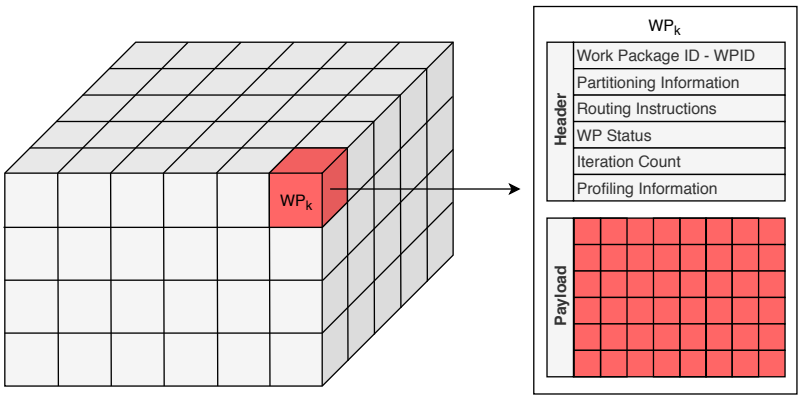


Figure 3.5.: Workload Partitioning: Work Package

gin of the original data domain. This is usually required by the OpenCL kernel where the computation is based on the spatial coordinates. Routing instructions are used to set the target node/device by the scheduler and is updated by the runtime for rerouting purposes when a device failure occurs. Work package status is an enum updated whenever a work package is conveyed from one stage to another, while iteration count keeps the number of iterations completed for iterative applications and applications with dependencies. Finally, device profiling information is stored by the computing component after each successful work package computation and conveyed to the scheduler (dispatcher).

In Table 3.1 we have shown some of the advantages of using work package approach for programming heterogeneous clusters. The main advantage of this work package approach is that achieving workload balance between cluster nodes is not entirely related to the partitioning sizes and the problem after partitioning shifts towards the work distribution strategy, which is more manageable. Partitioning is very complex with custom-size approach since it needs tuning of many hardware, software and application-centric characteristics to achieve workload balance between devices.

Table 3.1.: Complexity of handling different tasks with work package and custom-size approaches

task	**work package**	**custom-size data chunks**
partitioning & workload balance	medium	very complex
device failures	simple	complex
poor performance	simple	complex
optimizations for performance and energy efficiency	complex	extremely complex

Work package approach is superior also when it comes to handling device

failures and erroneous computations. Unlike in the custom-size partitioning approach, a cluster node discovering a device failure can reroute the work package to another device without having to deal with device capacity or other hardware limitations, since all work packages fit in all devices of the system. Further, it is also better at handling poor performance of devices. A non-performing device can be removed from computation and work packages rerouted to other devices. Besides the workload balance and handling failures, the work package approach excels at another very specific aspect. Since all work packages are the same size, we can measure the execution time and energy consumption of a work package for each device type of the system. With the profiling information, we can devise work distribution strategies that optimize for energy efficiency in addition to performance. This is extremely complex with the custom-size approach since the optimization problem becomes very complex and changing the size of the chunk affects both execution time and energy consumption.

3.1.2. Partitioning Strategies

In the previous section, we have dealt with partitioning workload from the perspective of hardware architecture by adjusting the chunks to ensure workload balance in asymmetric heterogeneous clusters. Another aspect that is crucial to the performance of data-parallel applications is the application structure. Not all datasets of an application can always be partitioned since data access patterns of an application can affect this process profoundly. Partitioning some datasets might transform applications in such a way that the intensity of data transfers or communications is increased to the point where it dominates execution and makes partitioning infeasible.

The execution process will be influenced depending on *what* we decide to partition. This decision might introduce synchronizations and barriers or communications between devices that are required to keep data coherency during the execution. Hence, from the programmer point of view, this decision amounts to a partitioning strategy.

Depending on application characteristics, and especially the data accesses of application's kernels, a decision has to be made whether it makes more sense to partition input, output, or both data domains. For instance, if the data access map is scattered over the input data domain for each output data point, perhaps it is more sensible to partition the output domain and replicate the input over all devices or vice versa. To illustrate this process, we present in Figure 3.6 the partitioning strategies for the DCS application, which we have discussed in Section 2.3.

DCS kernel has two datasets: input data domain which contains data about the atoms and output domain which reflect the spatial positions of the grid points and accumulate the influence of the electrostatic field in that point (see Figure 3.6a). There are three possible partitioning strategies, including partitioning only input, only output or both data domains. Which decision would

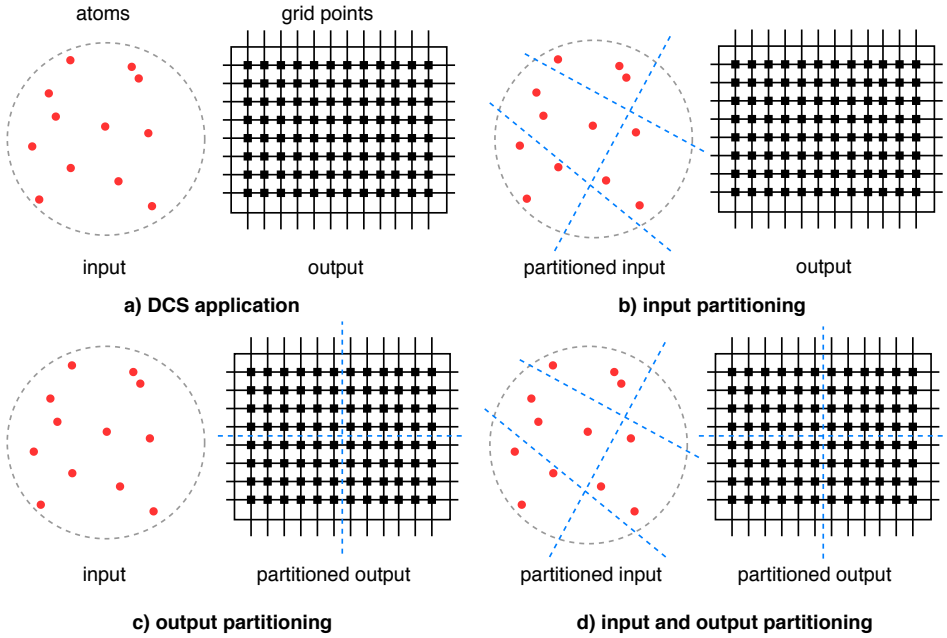


Figure 3.6.: Partitioning strategies for DCS

harvest more performance from heterogeneous architectures is hard to tell without analyzing the computational behavior of the application. DCS, for each grid point calculates distances from the grid point to atoms and computes the influence of atom's charges to that grid point. The partitioning strategy to be taken in this case depends on which amount is bigger, the amount of atoms or grid points while also the complexity of operations that are performed on data plays an important role. As shown in Figure 3.6b if the workload partitioning is based on input data, the output arrays would be replicated in all devices, while the work packages generated by input partitioning are distributed in different devices. The output arrays resulting from devices are aggregated at the end of the computing process. Conversely as shown in Figure 3.6c, if output array is partitioned, input arrays are replicated to the devices while output work packages are distributed in devices. Output work packages are finally merged at the end of the computing process. Partitioning of both data domains in this case is also possible (see Figure 3.6d).

For other applications that consist of many input and output data domains, the partitioning can be partial or complete in relation to input and output. In the case of partial partitioning, only a few or some of the input and/or output arrays are partitioned, whereas complete partitioning translates as either all input and/or output arrays are partitioned.

We will discuss the partitioning strategies for a broad range of HPC applications and in more depth in Chapter 5.

3.1.3. Spatial Partitioning

HPC applications usually consist of multidimensional datasets. The partitioning process retains the multidimensionality of the dataset, although it enables partitioning in a smaller number of dimensions than the original data domain. For instance, as shown in Figure 3.7 three-dimensional datasets can be partitioned in many ways: as smaller cuboids, as two-dimensional slices, two-dimensional blocks partitioning through columns and rows, two-dimensional blocks that include complete rows, two-dimensional blocks that include complete columns or as one-dimensional arrays.

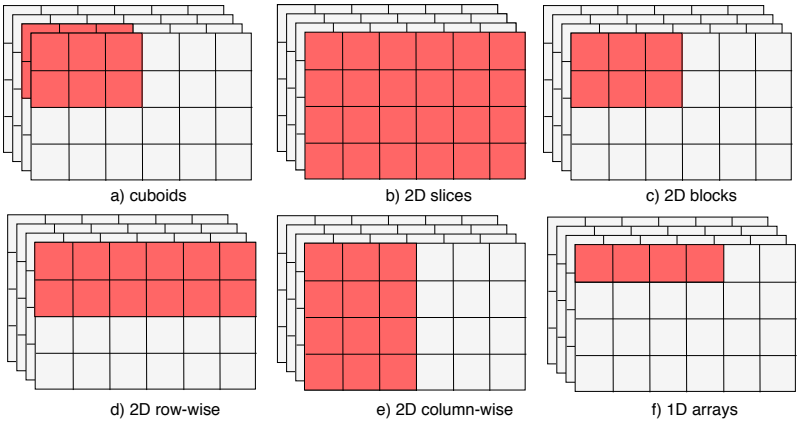


Figure 3.7.: Partitioning multidimensional arrays

Tuning Work Package Payload Sizes

An initial work package payload size can be tuned further to increase the performance of execution per work package. The initial probe (sample of work package payload) is tested in a selected number of devices of the cluster, each representing a device type. Profiling information is gathered which along with the execution time and data transfer times also contains power consumption data. Power consumption data are used when the scheduling strategy is a multi-objective optimization problem taking into account both execution time and energy consumption. Decreasing or increasing the size of the payload may result in performance gain in terms of overall execution performance.

Another issue is that different devices may often yield highly different execution times depending on computational behavior of the application and hardware characteristics. The tuner in this case aims to get a better balance in execution times between devices by testing a set of finite payload sizes while maintaining the workload balance between cluster nodes with scheduling strategies.

We experiment with many HPC applications within the scope of this thesis.

Experimental data with regards to tuning work package payload sizes are presented in more detail in Section 5.4.4. Our observations show that tuning work package payload sizes is highly beneficial, especially for certain types of applications such as imbalanced applications where the workload among work-items varies.

3.2. Work Distribution

Work distribution strategies are very important for accelerating executions and reducing energy consumption. We consider applications with a workload that is partitioned into work packages which are assigned to devices to be processed. Runtime efficiency and energy efficiency basically boils down to the problem of mapping work packages to devices. Different work distributing strategies are possible resulting in different solutions exhibiting different efficiency characteristics. An assessment of a solution heavily depends on the requirements of the programmer.

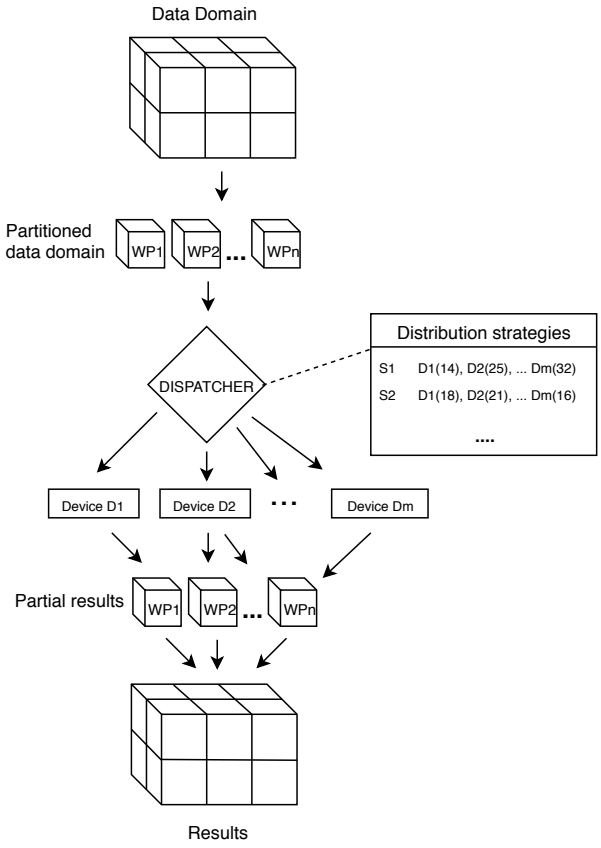
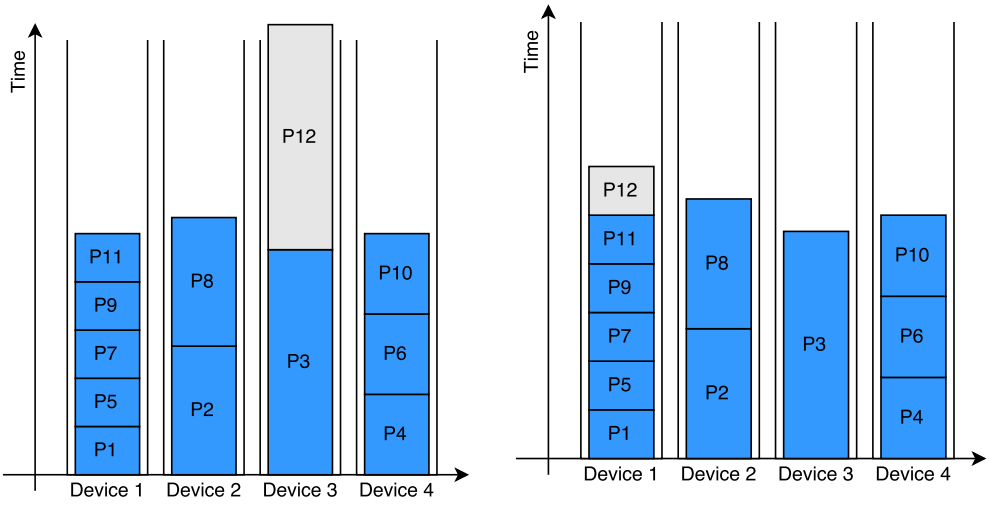


Figure 3.8.: Work package distribution

Usually, the number of work packages are orders of magnitude greater than the number of available devices in the system. This problem, coupled with different performance and energy consumption numbers for each of the devices leads to the requirement for a strategy in distributing work packages. Different types of mappings are realized by the *dispatcher* which manages the distribution of work packages. As shown in Figure 3.8, the dispatcher can perform different distribution strategies $(S1, S2, ...)$ which define different mappings; e.g. strategy $S1$ requires that 14 WPs are assigned to device $D1$, 25 WPs to $D2$, and so forth, whereas strategy $S2$ requires that 18 WPs are assigned to device $D1$ and 21 WPs to $D2$. The output is constructed from the partial results of the processed work packages.

In principle, to achieve the best performance all devices have to be used for computation. However, there are situations where this might lead to sub-optimal solutions. Consider a system as depicted in Fig. 3.9 with four devices with different processing times for work packages as indicated by the different lengths of the bars and 12 equal-sized work packages. In Fig. 3.9a, the work packages are distributed at runtime to the devices just on-request basis. Unfortunately, device 3 gets assigned the last work package 12, since the other devices are still busy.



(a) Simple request-based assignment

(b) More sophisticated assignment approach

Figure 3.9.: Different mappings of work packages onto devices

However, a better distribution of work packages exists, as shown in Fig. 3.9b. Although device 3 finished processing and is idle, it should not request another work package, but leave it to faster devices. Thus a simple on-request work package scheduler is not sufficient and more sophisticated techniques are required.

Further, optimizing distribution process by only taking into account the computing performance is not sufficient, considering that energy-efficiency is an important factor in heterogeneous systems. Hence, multi-objective optimizations of mapping decisions are necessary. We tackle this issue in more depth in chapters 6, 7 and 8 where we identify the optimization problems, provide efficient solutions and recommend trade-off solution of time and energy for mapping decisions.

3.3. Summary

In this chapter, we discussed in detail the problem of workload partitioning and distributing work onto devices of a heterogeneous, asymmetric cluster. We have presented our workload partitioning approach based on same-size medium-grain work packages, the advantages of which can be used to maintain workload balance, handle device failures, and poor performance. We analyzed strategies for partitioning HPC applications and discussed the importance of application characteristics in workload partitioning decisions. Moreover, we laid the groundwork to support optimizations for a broad range of distribution strategies taking into account both execution time and energy efficiency.

CHAPTER 4

clusterCL: A Framework for Heterogeneous Cluster Computing

The high variety of heterogeneous devices used for computation in nowadays high-end computers has introduced new challenges in programming such systems. The programmer is confronted with new issues arising in heterogeneous systems, such as partitioning and dispatching data to keep compute devices busy and achieve load balance, an issue easily handled in homogeneous systems. Heterogeneity is not limited to compute devices, but also includes cluster nodes with different hardware configurations leading to asymmetric cluster architectures. In such a hierarchical system OpenCL is not sufficient anymore. Support is required to distribute the work efficiently onto the non-identical cluster nodes. The different behavior of the individual compute devices with respect to execution time and energy consumption has to be taken into account to meet the demands of the user.

A hybrid programming approach with MPI and OpenCL complicates programming considerably, even in the context of symmetric nodes with identical hardware configurations. Typically MPI codes assume symmetric nodes resulting in an even distribution of work onto the nodes making it easier to ensure load balance. However, nowadays most of the heterogeneous clusters in computational labs are asymmetric with different kind of accelerator devices requiring new strategies in dealing with such systems. Achieving load balancing in heterogeneous asymmetric clusters is a complicated process that goes beyond simple data partitioning and requires a thorough analysis of hardware properties of the computing devices and examination of the code.

Moreover, considering only load balance and execution time is not sufficient anymore. In addition to performance, it is of key importance to take energy

consumption into account as well. It can be a reasonable strategy to exclude bad performing devices from execution to save energy without degrading the overall execution time substantially. In this chapter, we focus on the performance of the framework and deal with the problem of energy consumption only in extreme cases, e.g. in terms of excluding devices which consume a lot of energy. Optimization problems which consider both performance and energy consumption as part of a distribution strategy are analyzed thoroughly in Part III.

Our starting point is an OpenCL kernel code with parameter transfer specifications and partitioning information. The goal of the framework is to support partitioning and distribution of the work over the heterogeneous cluster nodes and to execute it without requiring the programmer writing MPI code or implementing dynamic work distribution strategies. The problems which have to be dealt within such an environment are manifold. The work has to be partitioned and data have to be copied to and from the compute devices. Since the communication links between nodes can be bottlenecks, prefetching of work is required to ensure a smooth operation of the devices. Smart scheduling and queuing strategies of work packages among accelerators are required taking efficiency considerations into account like execution time or energy consumption. This includes proper strategies for non-performant devices as well as handling of device failures with appropriate recovery actions to enable a successful program termination.

clusterCL provides a transparent view on the different compute devices alleviating the programmer from dealing with the hardware architecture and device execution behavior explicitly. Besides efficiency considerations, the device-sensitive feature includes in addition handling of device failures and appropriate recovery actions. Experiments show that our framework succeeds in distributing the work onto compute devices efficiently.

Contributions

Whereas most of the related work is concerned with OpenCL and solutions within cluster nodes, our framework targets the problems arising from a heterogeneous, asymmetric cluster environment. The main contribution of this work is to deal with these problems in an efficient way and to take from the user substantial programming burden when launching kernels in such a system. More precisely, the contributions are as follows:

(a) a framework that provides an abstraction layer to the programmer for the hardware and software architecture

(b) smart scheduling and queuing of work packages among accelerators taking efficiency considerations into account

(c) prefetching of work to hide communication overhead

(d) recognizing and dealing with non-performant devices and device failures

(e) showing the practicality of our framework, by increasing programmer productivity through eliminating some tasks such as coordinating dispatch or input/output reconstruction

In particular, our framework is novel in taking efficiency considerations into account, excluding automatically non-performant devices, and identifying device failures and performing corresponding recovery actions to enable successful completion of the computation.

Related Work

The related work can be classified into three main categories: (i) single device optimizations, (ii) single node frameworks, and (iii) cluster frameworks which are closest to our work.

(i) Single device optimizations: The objective of these research efforts is to optimize a generic OpenCL code for different, heterogeneous devices improving in this way portability. One of the first papers which discussed compiler transformations in the context of CUDA to support efficient GPU codes has been by Ryoo *et al.* [RRB+08]. Much work has been done for single device optimizations and the state-of-the-art has been pushed considerably. These research efforts can be seen orthogonal to our work, since we do not stress optimizations for individual devices, but address an efficient use of the whole cluster.

(ii) Single node frameworks: The objective of this research efforts is to provide support for devices located in a single node. Lutz *et al.* [LFC13] present a framework to optimize stencil computations for multiple GPUs on the same PCIe network. Similarly, Kim et al. [KKLL11] propose an OpenCL framework for multiple GPUs in a single node.

(iii) Cluster frameworks: One of the first efforts in this direction have been undertaken by Duato *et al.* [DPS+10] for CUDA and Barak *et al.* [BBNLS10] for OpenCL. A more recent approach is SnuCL [KSL+12] which enables viewing of remote devices as part of a local context while abstracting the communication between cluster nodes. libWater [GPCF13] is similar to the SnuCL approach, though it provides more efficient communication between cluster nodes, which is handled transparently by its runtime. Hybrid OpenCL [AONM11] is based on FOXC runtime and enables communication between different OpenCL implementations in the context of distributed computing, while our framework targets HPC clusters. dOpenCL [KSG12] allows using of different compute devices in any of the cluster nodes in a single application. It provides a central device manager which manages the assignment of the devices. clOpenCL [ARPS13] uses a wrapper library similar to dOpenCL targeting HPC clusters. DistCL [DGAJ13] is similar to our approach, as it intends to abstract the multiple-devices in a cluster providing the programmer a single-device view for

launching a kernel. However, DistCL runs its experiments across a symmetric cluster and only with one GPU per cluster, which is quite different from our asymmetric approach. VOCL [XBZ+12] provides a virtualization framework for GPU clusters using remote GPUs through proxy processes in cluster nodes. pVOCL [LLA+13] utilizes the VOCL framework and provides means for reducing energy consumption on a cluster. However, this approach is restricted to GPU clusters and it does not take into account actual energy consumption of applications, but it is based on a table power model which needs to be provided by data center administrators. Whereas the cluster approaches presented so far have their origin in OpenCL, HeteroMPI [LR06] is an extension to MPI to support heterogeneous networks of computers. HeteroMPI provides the functionality to enable an application developer to deal with a heterogeneous environment and to realize a required behavior, but does not support the programmer with advanced features.

None of the above-presented approaches are taking efficiency parameters in a cluster environment into account excluding automatically inefficient devices or recognize device failures and undertake recovery operations.

4.1. Framework for asymmetric heterogeneous clusters

In this section, we describe the architecture and the features of our framework.

4.1.1. Software architecture of our framework

Our software architecture is based on MPI, a thread library, and OpenCL. The communication and orchestration between the front-end node and cluster nodes is done via MPI. As shown in Figure 4.1 for each cluster node a pair of threads is created, handling concurrently MPI calls for sending work packages to the cluster nodes, *sendWP()*, and for receiving processed work packages from the nodes, *recvWP()*. Cluster nodes run OpenCL and steer all the local compute devices. For handling device requests promptly, a multithreaded approach has been realized: the cluster node runs for each compute device a thread and additionally one thread for MPI requests.

The key task of our framework is to distribute the work over the heterogeneous cluster nodes taking efficiency considerations automatically into account. Fig. 4.2 shows the three components of our framework with the main modules.

The *supervisor component* runs on the front-end node, whereas the *coordination component* and *computation component* are executing on the cluster nodes. Basically, the supervisor component steers the computation and distributes the work into the cluster nodes. This is done by partitioning the workload into same-size work packages and dispatching them to the cluster nodes. Thus we focus on application classes for which such partitioning is viable.

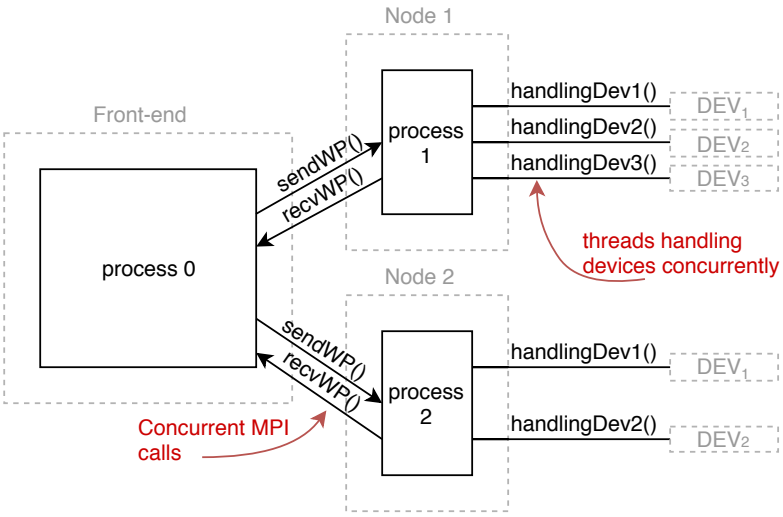


Figure 4.1.: Software Architecture

A prefetching mechanism has been implemented through the *prefetcher* module with the aim of hiding the communication between the supervisor component and the coordination component. Work packages received by the prefetcher module are handed over to the *coordination and deployment module* which distributes them to the computation components. The computation component is responsible for setting up all OpenCL resources and objects which are required for launching a kernel on a device realized by the *kernel launcher module*.

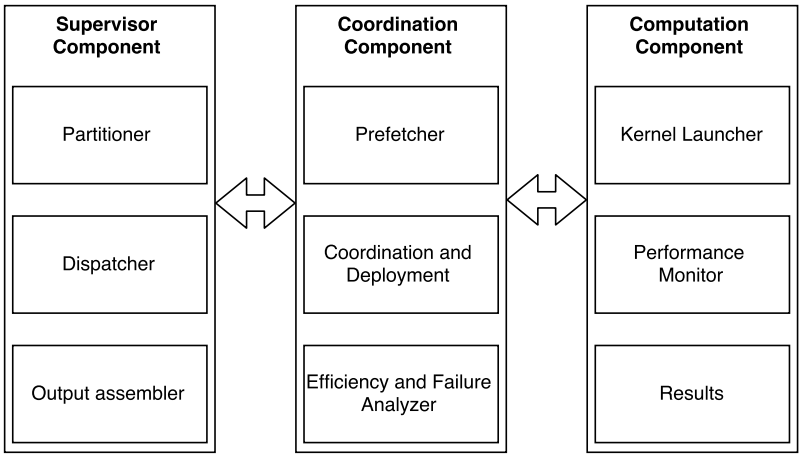


Figure 4.2.: Block diagram with main framework modules

Since the time required for some device to handle a work package is not known in advance, a tracking facility has been realized. The *performance monitor*

measures execution time and energy taken by compute devices to process work packages.

Another important feature of our framework is the handling of device failures. Resilience to faults is an important topic which has to be addressed with various strategies. There are many reasons for failures which can occur at different levels like at node level or at device level. The most common fault-tolerance technique is based on checkpoint and rollback recovery. Our failure handling does not replace the checkpoint-rollback-recovery strategy, but our framework helps to reduce the number of expensive recovery actions by handling device failures at nearly no additional costs. Device failures occur when an error is detected by the computation component during the kernel launching or execution process. Such failures are reported to the coordination component which terminates the corresponding computation component, releases all resources associated with it, and initiates recovery actions by reassigning unprocessed work packages to other compute devices.

Supervisor component

The supervisor component runs on the front-end, initiating the computation process on all cluster nodes. Its main function is to partition the workload and to dispatch the work packages to the cluster nodes. Figure 4.3 presents a detailed modular view of the supervisor component. Modules depicted with dashed lines require more in-depth analysis and are discussed and evaluated in more depth in separate chapters.

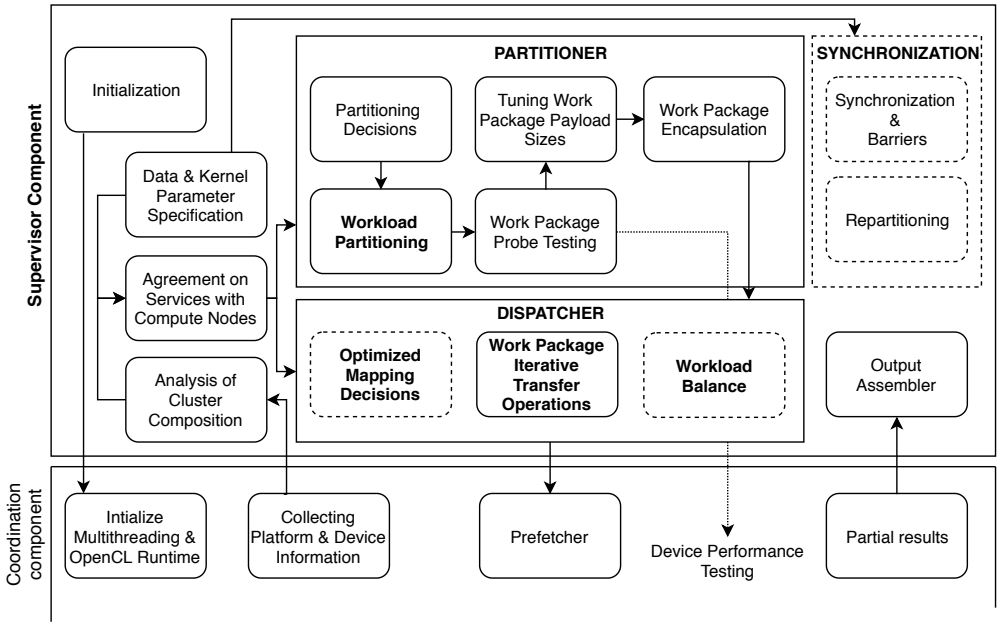


Figure 4.3.: Detailed modular view of the Supervisor Component

As an initial step, the supervisor component analyzes the data domain and the chosen workload partitioning strategy. The programmer is required to provide data domain specifications as outlined in Table 4.1. Kernel specification and sequence of kernel execution for multi-kernel applications is also analyzed at this step, where the programmer is required to specify the dependencies between kernels. Concurrently an analysis of the cluster composition is evaluated based on the data provided by the cluster nodes. The supervisor component builds an accelerator descriptor table summarizing the characteristics of compute devices available for computation in the cluster. Based on the gathered application characteristics and cluster composition, an agreement on services with cluster nodes is achieved delineating which devices will be used in the execution process and how the data will be transferred. This agreement is passed on to the partitioner and dispatcher module of the supervisor component.

Table 4.1.: Data domain characteristics

Data domain characteristics	**Description**
kernelArgumentID	Represents the order of parameters in kernel header
isArray	Determining if data are array or primitive values
dataType	Type of data to be processed
nDim	The number of dimensions for multidimensional datasets
sizePerDim	The length of data in each dimension
isPartitionable	Indicates if the dataset is to be partitioned or replicated
flag	Indicated the access rights to the data
inputOutput	Indicates if data are to be considered as input or output

The *partitioner* module runs an analysis guided by the entries of the descriptor table and the information provided by the programmer regarding the kernel structure. A partitioning decision is taken and initial workload chunks are created and submitted to representative devices for each device type. Since all devices of the same type are expected to process the workload the same way, the profiling information from the selected devices from each device type is sufficient for decision making. A small number of probes with different payload sizes is tested in the devices with the aim to decrease discrepancies in performance or energy consumption between device types. Finally, the payloads are encapsulated into work packages and conveyed to the dispatcher.

The role of the *dispatcher* module is to supply all cluster nodes with work packages produced by the partitioning process. It takes into account the decisions of the partitioner and decides about the distribution strategy. The default operation mode is a dynamic on-demand distribution strategy that can ensure workload balance. However, as we will see in the later chapters, the dispatcher relies on advanced mapping decisions which can produce optimized distribution

strategies that take into account performance and energy consumption at the same time. All work packages dispatched are tagged using unique numbers by the supervisor component to keep track of the submitted work packages. The descriptor table keeps information about the destination and status of each work package dispatched. The dispatcher module also receives feedback information from the coordination components of each cluster node about their status, e.g. when computing devices have been excluded from the computation process. This information enables the dispatcher module to update the accelerator descriptor table regarding the status of the compute devices and submitted work packages.

The *synchronization* module typically deals with intra-kernel and inter-kernel dependencies. Intra-kernel dependencies occur as a result of the partitioning process where kernels might require access to data which are not readily available in the device memory, while inter-kernel dependencies occur in-between kernel launching operations in the course of multi-kernel execution sequence. Chapter 5 tackles in detail the problem of dependencies and synchronizations for HPC applications and the impact of partitioning strategies in the execution process.

The result of the computation process has to be arranged by the supervisor component based on the partial results which are returned from the coordination component for each of the cluster nodes. All work packages returned contain the identification tag by which the supervisor component can check them and construct the output.

Coordination component

The *coordination component* runs on the host and constitutes the middle layer between front-end and compute devices, i.e. it realizes the interface to the front-end node and the interface to the compute devices. Its function includes receiving work packages from the front-end node, coordinating the deployment of work packages to the compute devices, and dealing with inefficient devices and device failures. Partial results of the computation are collected by the coordination component and delivered to the supervisor component. Figure 4.4 presents a detailed modular view of coordination and computation components.

In order to respond to device requests immediately multiple OS threads are created on the host: the host runs for each compute device one thread which runs the *computation component* and one additional thread for MPI requests. The coordination component initiates the OpenCL runtime on the cluster node and queries for available platforms and compute devices. It monitors their status and other characteristics used for work distribution decisions.

Each coordination component is required to identify the devices cluster node hosts and to probe their characteristics. Hardware characteristics are crucial for making workload partition decisions. A list of characteristics probed by the coordination component is given in Table 4.2.

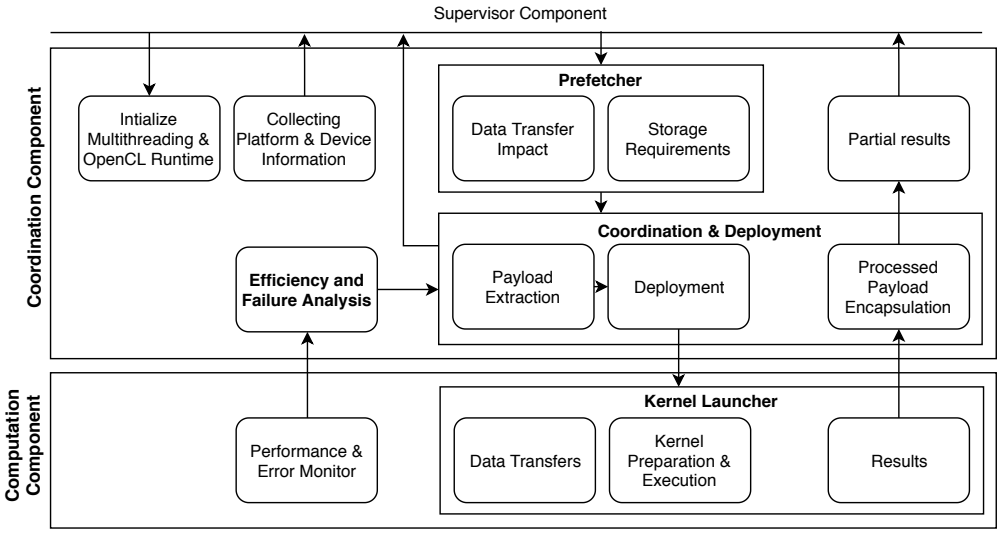


Figure 4.4.: Detailed modular view of the Coordination and Computation Components

A prefetching mechanism has been implemented through the *prefetcher* module, with the aim of hiding the communication between the supervisor component and the coordination component, by prefetching extra work packages while the devices are still busy processing. The prefetcher receives work-packages from the supervisor component. Since these work packages are stored locally, the prefetcher is responsible for negotiating the number of work packages sent by the dispatcher in a round. The amount is initially determined based on the computing capacity of the cluster node, while it is tuned afterwards by taking into account the collective performance of the compute devices.

The *coordination and deployment* uses OpenCL Runtime to discover platforms and create OpenCL device objects and contexts in the particular cluster node. The coordination and computation components use these objects for managing with other OpenCL objects created later on the process. Work packages received by the prefetcher module are handed over to the coordination and deployment module which distributes them to the computation components.

The performance of each compute device is measured and compared with other device types. If the performance of one accelerator is much lower than others, the *efficiency and failure analyzer* module proposes to the coordination and deployment module to exclude a device from the computation process. This decision is based on efficiency considerations given by the programmer. A similar procedure is applied in case of device failures. Decisions which exclude devices from the computation process are communicated to the supervisor

Table 4.2.: Hardware characteristics

Hardware Characteristics	Description
Number of Compute Units	The amount of streaming multiprocessors
Number of Processing Elements	The amount of cores
Global Memory Capacity	Size of the global memory
Local Memory Capacity	Size of the memory of streaming multiprocessors
Private Memory Capacity	Number of registers
Threading Capacity	The maximal number of threads that can be invoked
Work Group Sizes	The maximal number of threads per work group

component to adjust work package dispatching.

Computation component

Its basic function is to steer the kernel launching procedure and to collect the results of the computation process for each submitted work package.

The computation component is responsible for setting up all OpenCL resources and objects which are required for launching a kernel on a device. The first part of this component code consists of instructions which create objects that are used through the whole process lifetime and are not changed (kernels, command queues, buffers). The second part contains a loop which iterates over OpenCL commands for data transfers, kernel deployment and kernel execution. The loop stops after the coordination component signals stop of processing due to no more work packages received or abort due to inefficiency or failure. The result work packages are forwarded to the coordination component.

The *performance and failure monitor* module reports to the coordination component the performance data collected after each execution and any erroneous behavior.

4.2. Dynamic On-Demand Work Distribution and Handling of Device Failures

4.2.1. Prefetching mechanism

Communication and data transfers are performed at two levels: 1) between the front-end and the cluster nodes via InfiniBand/Ethernet and MPI, and 2) locally within a cluster node between the host and compute devices via PCIe and OpenCL. The communication volume is dominated by the data distributed to the devices. Above all, the communication between the front-end and the cluster nodes can be a bottleneck requiring special techniques to prevent performance degradation.

To resolve this problem, we have implemented a prefetching mechanism for work packages. While a compute device is busy with processing a work package, the request for the next work package is issued. The framework contacts the front-end and transfers the data such that they are locally in the node available when they are needed. In this way, the communication overhead is hidden by computation and idle time of devices is reduced.

The number of work packages stored in a node reflects the computational power of the installed devices. Due to asymmetric cluster architectures, the number of work packages kept locally by the prefetching mechanism may vary from node to node.

Local prefetching realizes the prefetching of work-packages from the host of a cluster node to the compute devices. It relies on the OpenCL out-of-order execution property of the command queues. When this property is set, OpenCL does not force the execution of the commands in a sequential order and further, it may execute the commands concurrently [Khr14]. Taking this into account, we have created two separate buffers in each of the devices, which are to be filled by the incoming work-packages. Based on the out-of-order property of the OpenCL command queue, our implementation assumes that the command for writing to a second buffer runs concurrently with the execution of the kernel which uses the data on the first buffer, ensuring prefetching of the incoming work packages in the device.

4.2.2. Dealing with poor performance and high energy consumption

Efficient execution of an application on a heterogeneous cluster is highly dependent on identifying and handling various performance obstacles. In our framework, an efficient execution is considered both in terms of execution time and energy consumption. Usually, these two quantities do not push in the same direction, since compute devices having a good performance often consume more energy and vice-versa.

Whereas measuring execution time is trivial, measuring energy consumption is more intricate and is done dynamically by querying the devices. Currently, major manufacturers such as NVIDIA and Intel provide APIs supporting queries for different device states including utilization, error status, energy consumption and other parameters. We calculate the energy consumption per work package and create a list of most energy-efficient devices based on the following formula:

$$E_{workpackage}[Ws] = P_{avg}[W] * T_{execution}[s] \tag{4.1}$$

The power draw of a device $P[W]$ is measured at different times during the execution of the work package in a compute device. Numbers from these readings are averaged when the processing of the work package is finished. Total

energy consumption in [Ws] is calculated combining averaged power draw P_{avg} and the execution time $T_{execution}$ of the work package. Based on these obtained values, we determine the energy efficiency of the devices for specific kernels.

The coordination component includes a mechanism which identifies devices with low performance or high energy consumption and excludes such devices from execution. Let us first discuss identifying such devices in more detail followed by a paragraph which addresses the exclusion of devices from the execution process.

Identifying Inefficient Devices for Exclusion

We distinguish between two different reasons for device exclusion:

1. poor performance, or
2. high energy consumption.

Both reasons are discovered by the coordination component by analyzing performance data and comparing them with framework parameters. In default operation mode with dynamic on-demand distribution strategy (no advanced mapping decisions) framework uses two parameters to make decisions on performance and energy consumption: one external parameter visible to the programmer (*performance parameter*) and one internal parameter controlled by the framework (*energy-efficiency parameter*).

When the computation process starts, work packages are assigned to the computational devices and runtime measurements are performed which are a prerequisite for our efficiency analysis.

For performance analysis, the programmer is requested to set the performance parameter which specifies a relation between the measured execution times. The parameter is used as a threshold for realizing device exclusion. Consider Fig. 4.5 with work packages P1 to P7 that have been assigned to a cluster node with 4 compute devices and 3 different device types. For instance, in Fig. 4.5 the parameter is set to 4 which means that a device executing a kernel four times slower than the fastest device shall be excluded from future computation for that kernel. Since device 1 is identified as the fastest device with execution time t_1 and has already processed four work packages while device 4 is still processing the first one, device 4 is labeled as non-performant and is a candidate for device exclusion.

On the other hand, the most energy-efficient version can be determined by our framework using the energy-efficiency parameter which is based on the execution time for a work package and the measured power draw. By taking formula (4.1) into account and identifying the most energy-efficient devices, the framework will decide on the primary device to be used for the rest of the computation. Although using only one device type for computations may decrease the degree of heterogeneity in the cluster, the system still remains

heterogeneous unless the most energy-efficient device is the CPU which means that no accelerator is used at all.

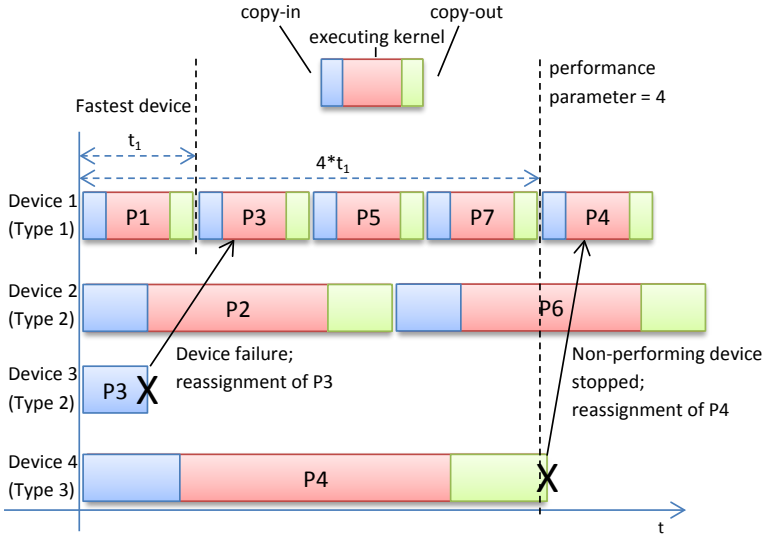


Figure 4.5.: Example for device exclusions and recovery actions

Realizing Exclusion of Devices.

If a device has been labeled as exclusion candidate, the following actions are performed. First, the computation component of that device is terminated and all resources and objects associated with it are released. Second, the unprocessed work packages are reassigned to other compute devices by the coordination component. Additionally, when the decision for the exclusion of a device from the computation process is taken, the prefetching mechanism is instructed to adapt the number of locally stored work packages to the new configuration.

4.2.3. Handling of Device Failures

Nowadays, resilience to faults is an important topic which has to be addressed with various strategies. There are many reasons for failures which can occur at different levels like at node level or at the device level. The most common fault-tolerance technique is based on checkpoint and rollback recovery ([You74, Dal06]): the application state is saved periodically throughout execution, and when a failure occurs, execution can be resumed from the most recent checkpoint.

Our failure handling does not replace the checkpoint-rollback-recovery strategy, which can handle successfully e.g. node failures, but our framework helps to reduce the number of expensive recovery actions by handling device failures at nearly no additional costs. Device failures occur when an error is detected

by the computation component during the kernel launching or execution process. This includes OpenCL runtime errors and problems that may be reported through OpenCL events mechanism when queried for a specific command. In Fig. 4.5 a computation failure occurs for device 3 for work package 3. This failure is reported to the coordination component which terminates the corresponding computation component, releases all resources associated with it, and initiates recovery actions as described above.

A special case of device exclusion occurs when a computing device does not respond after the kernel has been launched for execution and no performance parameter has been set by the user. For such non-responsive devices, it is unclear whether they are very badly performing or some hardware failure occurred. These cases are discovered by the coordination component when all the work packages submitted for computation have been processed except some long-running cases. In order to recover from such situations, these devices are excluded from computation and the work packages are reassigned as well.

4.3. Experiments and discussions

In this section, we present the results from our experiments regarding the efficiency of our framework and the strategies for handling non-performant devices. Further, we evaluate and compare clusterCL with a hand-coded implementation in a broad range of features.

We have tested our framework against our two distributed-memory clusters PHIA and CORA. PHIA consists of 8 nodes. Each of the nodes has two Intel Xeon E5-2650 CPUs with 8 cores, two NVIDIA Tesla K20m GPUs and two Intel Xeon Phi 5110p co-processors. Nodes 7 and 8 have different configurations hosting four NVIDIA Tesla K20m GPUs and four Intel Xeon Phi 5110 co-processors, respectively. CORA consists of 7 nodes, each of them containing two NVIDIA Tesla C2050 and one NVIDIA Tesla C1060 along with two Intel Xeon quad-core X550 CPUs.

Our initial experiments are based on the analysis and visualization of a molecular dynamics code. In order to prove the usability of our framework in distributing the workload efficiently among cluster nodes and compute devices, we have experimented with various problem sizes of the OpenCL kernel and different hardware configurations. In the second part where we compare clusterCL to a hand-coded implementation, besides DCS, we use a stencil routine, Alternating Direction Implicit (ADI) and two synthetic data-parallel routines, the first a data-intensive application and the second a compute-intensive application.

4.3.1. Evaluation of clusterCL

In this section, we show the experimental results of the evaluation of various aspects of clusterCL. We discuss the efficiency of our approach by showing

how well it scales in our clusters. Further, we evaluate the effects of workload distribution and the effects of prefetching. Last part evaluates the impacts of device exclusions in the execution process.

Efficiency

In this experiment, we study the performance numbers when switching from a single node solution to a multi-node cluster. Since we want to examine the efficiency of our framework, we used two different scenarios in our PHIA cluster: 1) allocating identical cluster nodes (experiments with nodes 1-6) and 2) allocating cluster nodes with asymmetric internal configuration (experiments including nodes 7 and 8).

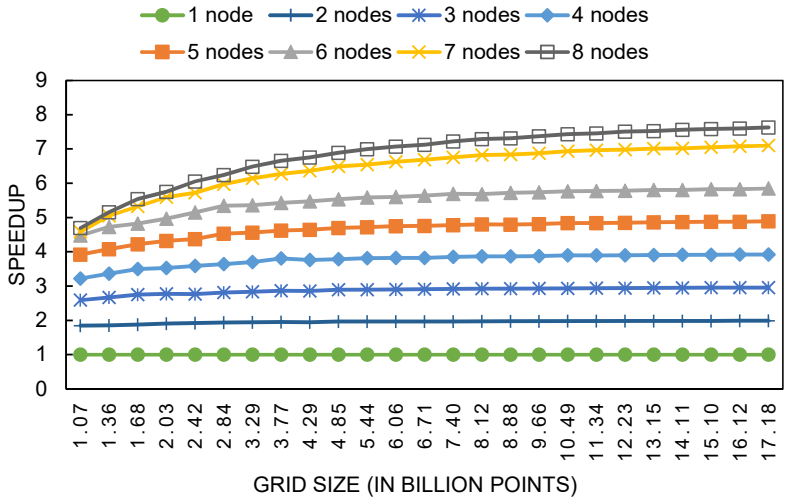


Figure 4.6.: clusterCL: Speedup in PHIA cluster

For both cases, a single node execution time is used as a baseline for the speedups. In our CORA cluster we experimented only with the scenario with identical cluster nodes. As shown in Fig. 4.6, our framework runs efficiently in a multi-node environment. The speedup corresponds almost to the number of nodes used for computations and it gradually increases with larger problem sizes. In the experiments where we have used 7 nodes of the PHIA cluster (6 nodes with identical configuration and the 7^{th} node consisting of four NVIDIA GPUs) the speedup is 7.1 times, resulting from the higher performance of the node 7 compared to other nodes with mixed configurations. This is also confirmed in the experiments with 8 nodes where the performance of the 8^{th} node is lower than that of other nodes, resulting in slightly lower overall performance than observed in other scenarios.

Running the computations with small problem sizes like $32k \times 32k$ on PHIA cluster may delay the completion of the overall execution, since there are only a small number of work packages to be dispatched. This problem also affects the

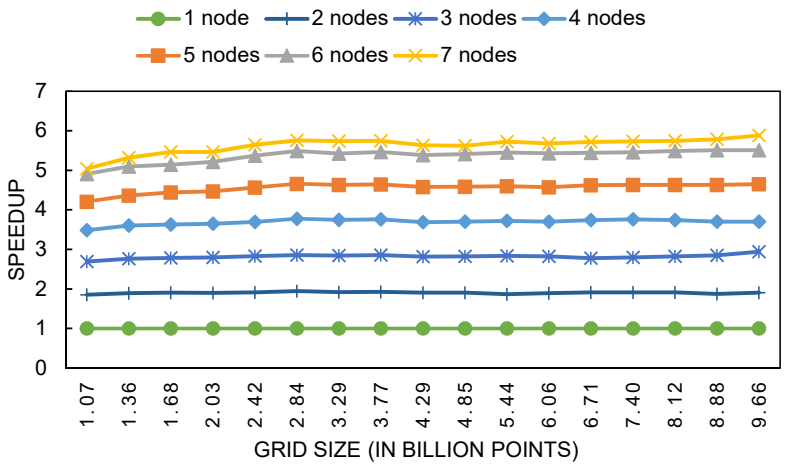


Figure 4.7.: clusterCL: Speedup in CORA cluster

CORA cluster as shown in Fig. 4.7, but for the same problem size, the supervisor component produces much more work packages of smaller sizes, based on the compute capabilities of the devices hosted in CORA.

Data and workload distribution

This experiment is designed to show the differences in work distribution between an asymmetric-aware and an asymmetric-oblivious strategy. The former, partitions data and workload in smaller chunks (work packages) and supplies cluster nodes on-request basis. The latter involves partitioning of data and workload in equal sizes between the cluster nodes at the beginning of the process. Our experiments are done on PHIA hardware.

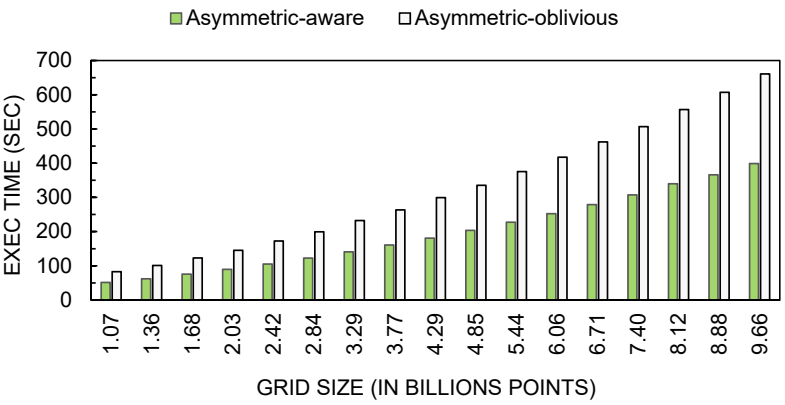


Figure 4.8.: Effects of the dynamic workload distribution

As shown in Figure 4.8, there is a big difference in the overall execution time

between asymmetric-aware and asymmetric-oblivious distribution strategies in asymmetric clusters. The solution with which does not take into account the asymmetric configuration of the cluster is taking up to 40% more time to complete for all problem sizes. This comes as a result of loading all the nodes with the same amount of work and data without taking into account the performance of the devices used.

Prefetching

In this experiment, we investigate the impact of the use of prefetching on communication hiding.

As shown in Fig. 4.9, in the experiments using the prefetching mechanism the amount of time required to transfer data from the front-end to cluster nodes (*transfer cost*), relative to the overall execution time, is significantly lower in comparison to the experiments not using the prefetcher. The transfer cost shown in the experiments using the prefetcher comes mainly as a result of the administrative overhead of thread management in the host and for setting up OpenCL resources and objects.

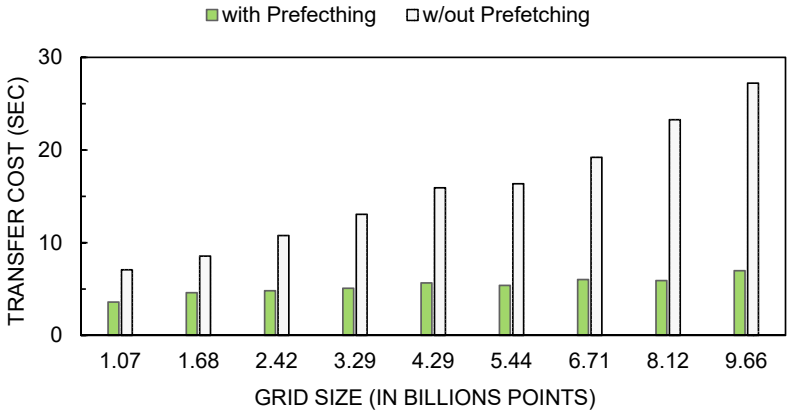


Figure 4.9.: Communication hiding using the prefetching mechanism in the cluster nodes

The contribution in communication hiding of the local prefetcher between the host on cluster nodes and compute devices for the applications such as analysis of molecular dynamics is negligible in comparison to overall application execution time. However, the local prefetcher may contribute significantly in reducing transfer cost for kernel types which have a more balanced ratio between the transfer and execution time.

Device Exclusion with Performance Parameter

In this experiment, we investigate the effect of setting the performance parameter to different values to exclude non-performant devices. The performance parameter allows the programmer to find a compromise between execution time and energy consumption.

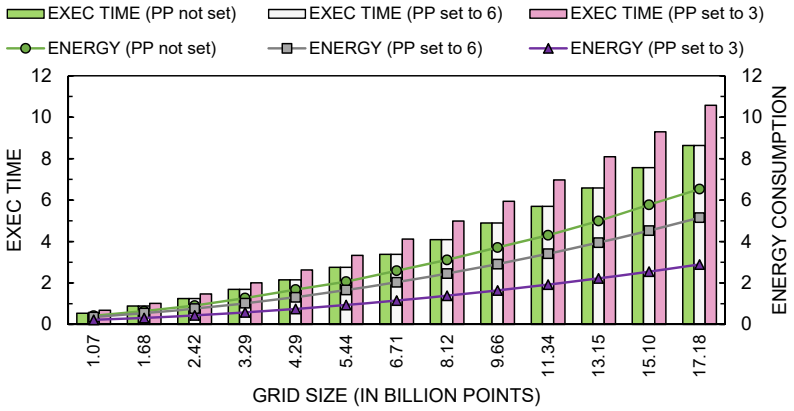


Figure 4.10.: Impact of the exclusion of devices from computation in overall execution time and energy consumption

As shown in Fig. 4.10, when the performance parameter is set to 6, the worst performing devices in our cluster are excluded. In this way, the energy consumption can be reduced considerably, however with the consequence that usually execution time will increase. In this case, the performance did not degrade more than 3% for any of the problem sizes, while the energy consumption was reduced by 20%. For small problem sizes, we have observed even better performance. However, that is because of eliminating the delay caused at the end of the execution by the worst-performing devices. When excluding devices that perform in the range between the best and the worst, i.e. by setting performance parameter to 3, our measurements show that the overall execution time is increased by 20%-25%, while the energy consumption was reduced by 50%.

4.3.2. Comparison of clusterCL to a hand-coded implementation

In this section, we present the experimental results of the comparison of clusterCL and a hand-coded implementation for DCS, ADI, and two synthetic data-parallel routines: DIR - data-intensive routine and CIR - compute-intensive routine. For each application, we realize two versions and compare them: a work package (WP) approach based on our framework version and a hand-coded non-WP version. Such a comparison is not so straight-forward and several assumptions have to be made. For instance, the problem size of the hand-coded

version is limited by the available memory, whereas the WP version supports bigger problems as well. Thus bigger problems can not be used for comparisons sacrificing an advantage of using our framework.

Computing Performance

In this section, we show the computing performance of the framework implementation (FWP) and a hand-coded implementation (HCI). These measurements include: the overall execution time (OET) - where we measured the time between MPI initialization and finalization; device computing times (DCT) - measuring the time of writing and reading data to and from the device and kernel execution time; communication time (CMT) - which includes measurements of time spent for communication between front-end node and compute nodes and between compute nodes; and setup time (SET) - which measures the time required for reading data from file on the front-end node and OpenCL Runtime initialization on the compute nodes.

Experimental results presented in Figures 4.11 - 4.14 show that the framework implementation is superior in terms of overall execution time in comparison to the hand-coded version for almost all applications. This is even more evident for bigger problem sizes where the gap between the overall execution times of two versions is prominent and gets wider continually. On the other hand, as shown in all cases, the framework version based on the WP approach competes with or lags behind the hand-coded version for small problem sizes. This comes as a result of a trade-off where the framework decides on the number of WPs produced taking into account their sizes to preserve the estimated optimal performance of the devices. Since the problem size is too small, the framework produces a smaller number of WPs, not enough to keep the devices busy all the time. This helps in hiding the communication between compute nodes and front-end node occurring in parallel with computing process as observed for bigger problem sizes.

DCS. Experimental results for DCS are depicted in Figure 4.11. These measurements show that the framework implementation is performing better than the hand-coded version for all problem sizes. According to these results, the framework implementation performs 30% faster than the hand-coded version for the biggest problem size analyzed. In the case of DCS this holds even for smaller problem sizes since the DCS code is a compute-intensive code. Therefore, longer device computing times compensate for smaller number of WPs produced by the framework.

For this class of applications, the communication time between cluster nodes and the setup time are similar for both implemented versions. The reason behind frameworks better performance in computing lies with a better estimation of the WP sizes. This feature of the framework induces better workload balancing between computing nodes compared to the equally partitioned data chunks

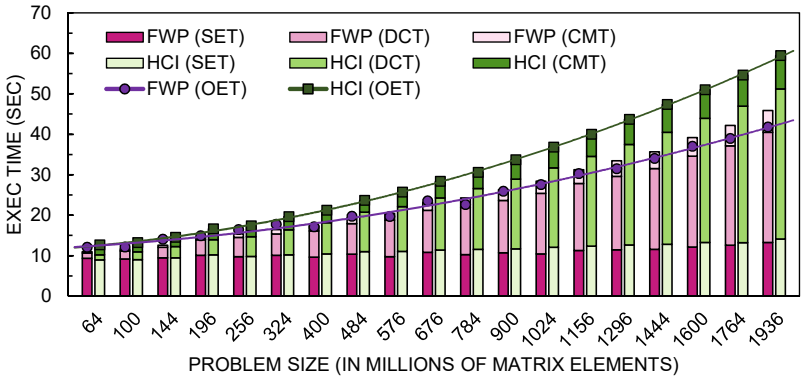


Figure 4.11.: Evaluation of performance of the framework and hand-coded implementation for DCS

of the hand-coded version.

As shown in the Figure 4.11, the overall execution timeline is projected lower than the combined times of setup, computing and communication in the case of the framework implementation. This results from communication hiding realized by the prefetching mechanism of the framework. While the combined times show independently measured times for each of the processes, the overall execution time does not account for the communication performed in parallel while the devices are busy executing other WPs.

ADI. Next, we have experimented with ADI. Experimental results are shown in Figure 4.12. These experiments show that the framework implementation performs slightly better than the hand-coded version.

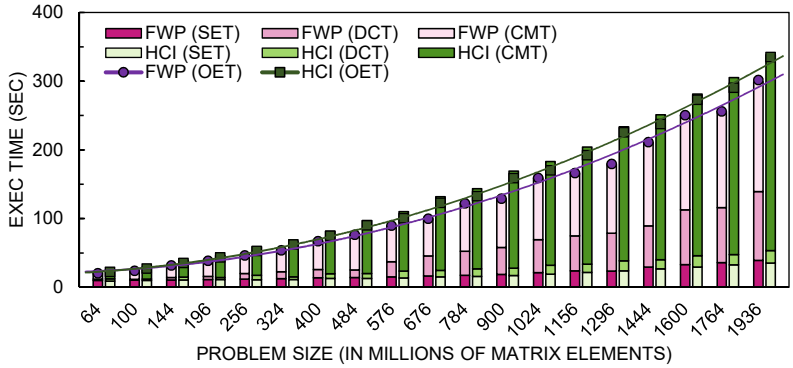


Figure 4.12.: Evaluation of performance of the framework and hand-coded implementation for ADI

The overall execution timelines follow the same pattern as in the case of DCS for smaller problem sizes. For bigger problem sizes, in the case of the hand-coded version, the communication dominates during execution. A

different situation is observed with the framework implementation, where the computation gets a larger share of the overall execution time. This is an ADI-specific issue related to its code structure. Since the partitioning of data is done row-wise, the number of threads created by the process is equal to the number of rows, with many WPs created for bigger problem sizes. Considering that the computational process sweeps along the rows, the computing time is related to the length of rows. Thus the difference in computing time for smaller data chunks such as WPs and bigger data chunks created by the hand-coded version is not significant. This leads to a much larger accumulated computing time in the case of WPs. However, since most of the WP transfer occurs while the devices are busy computing, the overall execution time is not increased proportionally.

DIR and CIR. Figure 4.13 and Figure 4.14 present the computing performance for DIR and CIR types of applications.

In the case of DIR, the overall execution time is dominated by the communication time. Considering that the framework communicates back all the WPs to the front-end at each iteration, while the hand-coded version performs only the boundary data exchange, the latter outperforms the framework version for all problem sizes. However, the performance gap, in this case, is much narrower than what can be observed in experiments with other application classes where the framework version performs better.

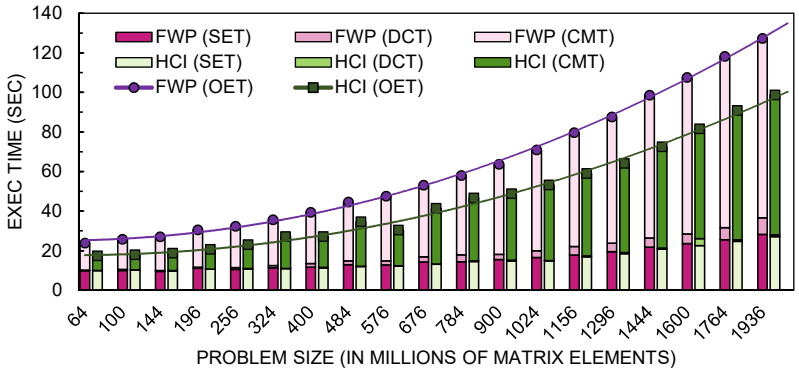


Figure 4.13.: Evaluation of performance of the framework and hand-coded implementation for DIR

Experiments with CIR show the opposite. The framework version outperforms the hand-coded version and the performance difference between the two versions increases with the increase of problem sizes. Our experimental observations show that the framework version performs 48% faster than the hand-coded version for the biggest problem size analyzed. Since this type of application is compute-intensive, the framework is capable of hiding most of the communication between the front-end node and compute nodes. On the other hand, the

hand-coded version struggles with synchronization between compute nodes resulting from the imbalance of workload due to heterogeneity and asymmetrical composure of the cluster.

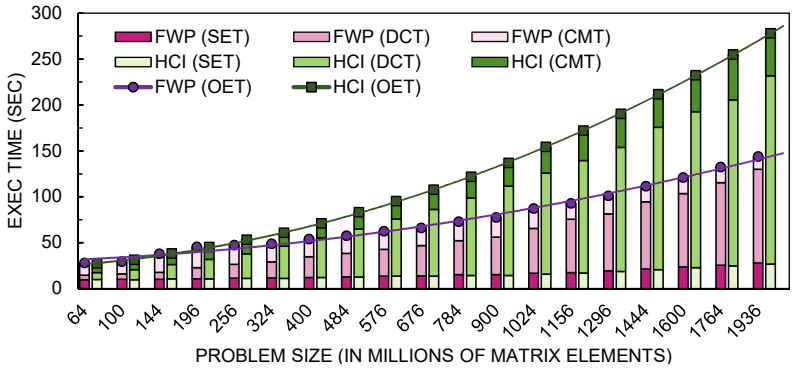


Figure 4.14.: Evaluation of performance of the framework and hand-coded implementation for CIR

Scheduling Performance

In this section, we look at the scheduling performance, i.e. the comparison of the implementation versions in their ability to utilize the computing devices. This experiment examines ratios of computing time and communication time to overall execution time, excluding the setup time which is similar in both implementation versions. Fig. 4.15 shows those ratios for the framework implementation for all the routines, while Fig.4.16 for the hand-coded implementation.

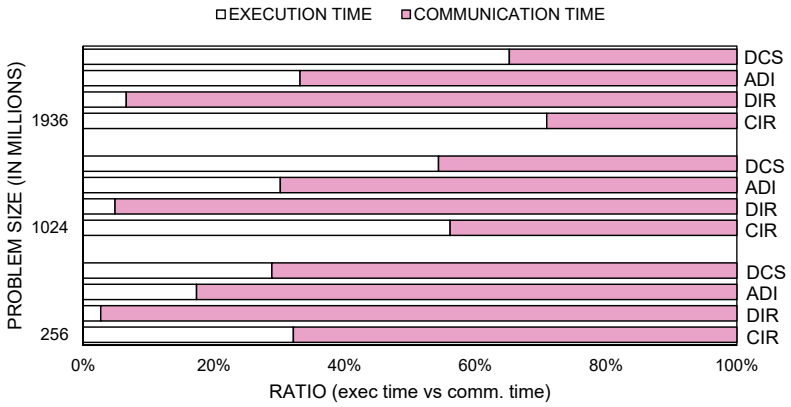


Figure 4.15.: Computing and communication time during computation process: Framework Implementation

The most prominent characteristic that can be observed in this experiment is

the steady increase of the execution time ratio in the framework implementation with the increase of the problem size. For the hand-coded implementation, an increase of the computing time ratio to overall execution time can be observed for smaller problem sizes. However, this ratio remains unchanged with bigger problem sizes, contributing to an increase in overall execution time.

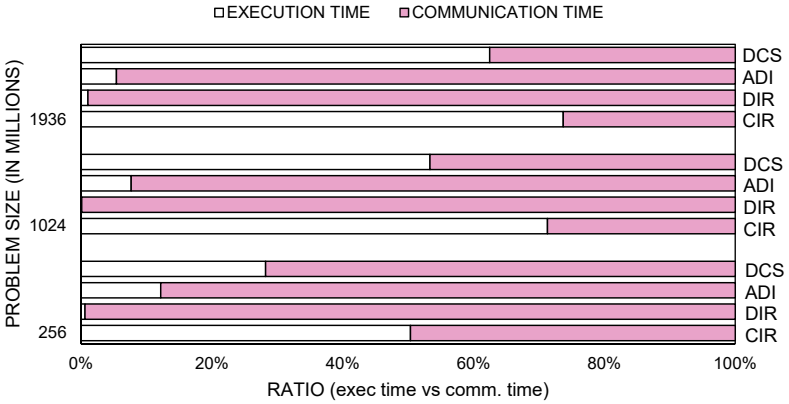


Figure 4.16.: Computing and communication time during computation process: Hand-coded Implementation

4.4. Summary

In this chapter, we have presented a framework which supports programmers to port single device OpenCL codes to multi-node clusters without requiring writing MPI code or dealing with distributing the work onto the nodes explicitly. The framework is novel in providing optimizations towards execution time or energy consumption, excluding non-performant devices or devices with high energy consumption. Moreover, device failures are recognized and recovery actions are undertaken to enable successful completion of the computation.

CHAPTER 5

Programming Support For Generic Data Parallel Applications

Programming heterogeneous devices is enabled and simplified by a few programming models such as CUDA [nVi19a], OpenCL [Khr14], OpenACC [Ope19a] and OpenMP $\geq$ 4.0 [Ope19b]. Barring CUDA, other programming models offer a uniform programming approach to various types of compute devices. Basically, a data-parallel kernel written for a device can be easily ported to any other compute device that supports the original programming model. However, as we have seen in the previous chapter, all of these programming models are only partially useful when the performance gain is sought beyond the optimization strategies for single device units.

In heterogeneous computing, the computational behavior of the application is another obstacle. Apart from the embarrassingly parallel applications, most of the data-parallel applications are very hard to partition. They have memory access patterns or execution paths that require data exchange between devices or synchronizations as a consequence of the application partitioning.

Several studies [LSPM13], [LSM15], [GPCF14], [SVL+16] have proposed approaches on how to provide programming support that enables execution in cluster settings or deal with workload partitioning in such systems. However, none of them engage in a wider perspective of handling complex scientific applications or ensuring workload balance in heterogeneous asymmetric clusters which are a common setting in scientific laboratories.

In this chapter, we analyze the strategies which tackle these applications and discuss implementation details of clusterCL which offer broader programming support for complex HPC applications. With these extensions, clusterCL is capable of handling a broader class of regular data-parallel applications and

a more complex hardware configuration of cluster systems. We extend *clusterCL* to coordinate the execution in a cluster node between cluster nodes and compute devices while steering the computing process for single or multi-kernel applications by handling the synchronizations, dependencies or data restructuring induced by the partitioning process.

Contributions

Whereas the base version of clusterCL discussed in the previous chapter is capable of handling simple single kernel data-parallel applications, extensive programming support is required when dealing with more complex HPC applications. Dependencies between work packages as well as dependencies between kernels in multi-kernel executions both occur as a result of workload partitioning and the partitioning strategy chosen by the programmer. Our main contribution within the scope of this work is extending clusterCL to support the partitioning process and provide data coherency for the execution of complex HPC applications. More precisely, the contributions of this work are:

(a) extensive programming support for seamless computing of data-parallel applications

(b) classification of applications by taking into account partitioning strategies, data access patterns and dependencies

(c) efficient handling of communications and synchronizations stemming from partitioning of parallel applications with dependencies

(d) minimizing device idle time by overlapping execution with data transfer

(e) performance improvements from the tuning of partitioning sizes

Related Work

Many studies have shown the benefits of providing programming support for heterogeneous cluster architectures in terms of performance gain and increased programmer productivity [KSL+12], [ARPS13], [DGAJ13], [KSG12], [GPCF14], [AONM11].

rCUDA by Duato et. al. [DPS+10] is among the first frameworks that have been developed to support multi-device programming for nVidia GPUs. Similar work based on OpenCL and extending the diversity of device platforms have been proposed by Barak et. al. [BBNLS10]. Alves et. al. [ARPS13] propose a platform clOpenCL that enables deployment of simple OpenCL based applications to different devices in a cluster environment. dOpenCL [KSG12] provides a central device manager which manages the assignment of kernels to devices. Similarly, Diop et. al. [EA12] developed DistCL to support distributed execution of OpenCL kernels in a GPU cluster. Aoki et. al [AONM11] have implemented a framework which consists of a runtime that provides an

abstraction layer for OpenCL implementations and a bridge program to connect those implementations. SkelCL [SKG11] is a library providing algorithmic skeleton that aims at alleviating programming GPU systems. All of those approaches have proved that enabling distributed execution of OpenCL kernels is important and valuable. They are focused in providing an abstraction layer to the programmer, without dealing in detail with intricacies of the data-parallel applications. Our approach is an essential improvement to these studies since clusterCL includes advanced features beyond simple OpenCL kernel deployments to the devices in cluster nodes.

Other works have tackled the problem of extending OpenCL platforms to clusters using language extension to support communication via MPI [KSL+12], [GPCF14] and recently [RBW+19]. In [KSL+12], Kim et. al. provide a framework implementation, SnuCL, which enables programmers to view all of clusters devices as if they are in the host node. However, the programmers should be aware of simple MPI calls in order to transfer data between nodes. In a similar fashion, Grasso et. al [GPCF14] provide a library, libWater, which handles transparently communications and data transfer between compute nodes, besides enabling distributed kernel execution to the devices. Rasch et. al [RBW+19] developed a high-level C++ library which simplifies the development of host code and provides optimizations for data transfer and memory management. Although these works are very similar to our approach of providing an abstraction layer for the programmer, they differ fundamentally from our framework, having that clusterCL require only parameter specification for OpenCL kernels, while the process of partitioning, distribution and scheduling of kernels and work packages is handled automatically by the framework ensuring workload balance between cluster nodes.

In [SVMS15] Shen et. al. propose a multi-kernel partitioning approach for heterogeneous single-node systems. Their work is closely related to ours in terms of the partitioning of work for multi-kernel applications and dealing with dependencies between kernels. However, our approach is different in many aspects. Whereas in [SVMS15] they use a profiling and prediction technique to determine the best work partitioning for specific devices, we use the work package approach of fine-grain same-size data chunks and improve the overall performance from distribution of work packages to the devices. This, in turn, enables us to perform work package rerouting in case of device failures while with custom-size partitioning this is much harder to achieve and sometimes impossible. Further, our approach extends to clusters and our framework extends the optimization model to minimize the data transfer costs incurred from inter-node communications.

The main difference of our approach and the works cited above is that clusterCL is a framework with many advanced features providing comprehensive support for workload distribution for a broad range complex data-parallel applications in heterogeneous asymmetric clusters.

5.1. Comprehensive Support for Partitioning-Induced Dependencies of Data-Parallel Applications

In the base version of clusterCL [RM15], we have shown how clusterCL enables execution of single kernel HPC Applications, which generally require no data transfer between the devices, a class of data-parallel applications similar to a bag of tasks applications. In this section, we extend our approach to support work partitioning and distribution for more complex data-parallel applications. We have examined many HPC routines and real-world HPC applications in order to come up with some general characteristics of applications.

Let us first discuss a general case of multi-kernel execution process using clusterCL. Usually, data parallel programs consist of a few kernels which are executed in a serial fashion, different from task-parallel programs which frequently have a much more complicated execution sequence. Lets assume we have an application with two kernels $k_1(a, b)$ and $k_2(c, d, e)$ where a, c are input non-partitioned (replicated) arrays for the kernels, d is a partitioned input array $d_1, \cdots, d_n$ which is modified by kernel k_2 and b, e are partitioned output arrays $b_1, \cdots, b_n; e_1, \cdots, e_n$, respectively for kernels k_1 and k_2. We assume that work has been partitioned into n work packages and that the cluster hosts m devices. Number of work packages is bigger than the number of devices, $n >> m$.

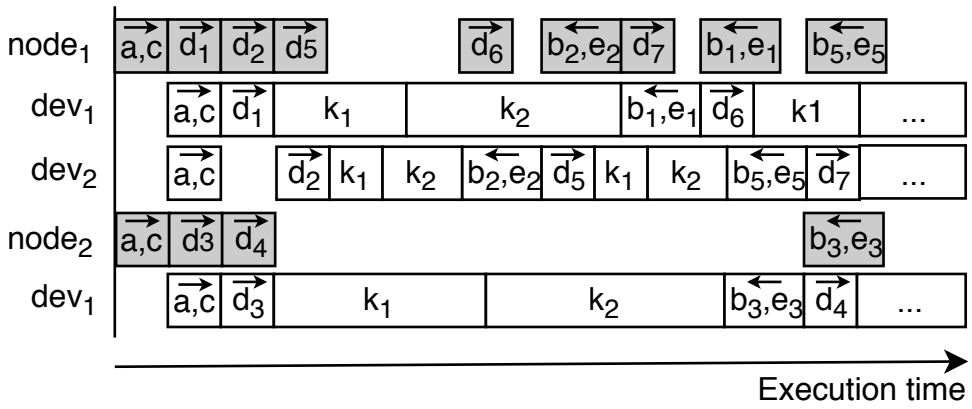


Figure 5.1.: Multi Kernel Execution with clusterCL

Figure 5.1 illustrates the execution process. There are two nodes, with the first node having two devices and the second node has one device. Data transfers are marked with arrows. With $\rightarrow$ we have marked data transfers to nodes or devices while with $\leftarrow$ are marked data transfers from devices to nodes or to front-end. Darker rectangles represent data transfer from front-end to cluster nodes and vice-versa, while lighter rectangles containing arrows represent data transfers from cluster node to the device and vice-versa. Execution process of

work packages in devices is marked with rectangles that contain no arrows. k_1 and k_2 depict execution of respective kernels in devices.

As shown in the figure, first we transfer non-partitioned input arrays a, c which are replicated to all devices. This is followed by the partitioned input array d in the form of work packages which are transferred on-demand basis or using a scheduling algorithm. Finally, computed output arrays b, e are transferred to front-end in the form of processed work packages. For the sake of simplicity, the figure shows the inbound and outbound data transfers as serial, although clusterCL implementation is configured to support *MPI_THREAD_MULTIPLE* for cluster-level communication which enables concurrent MPI calls, and out-of-order queuing and execution to enable overlapping of data transfers from nodes to devices with kernel execution. We can observe that device dev_2 is the fastest, while device dev_1 of the second node, the slowest. Using the prefetcher, a few work packages are stockpiled on each node, which ensures smooth operation and minimizes device idle time.

In the course of execution of this application, two issues of great importance have to be handled:

- *intra-kernel dependencies*
- *inter-kernel dependencies*

With intra-kernel dependencies we define data dependencies resulting from kernel data access requirements after partitioning, e.g. kernel k_2 processing work package d_1 in device dev_1 of $node_1$ requires access to data packed in work package d_3 which is processed by dev_1 in $node_2$, or it requires access to data in work package d_8 which hasn't been dispatched yet by the front-end. In both cases, data are not readily available in the dev_1 memory. This computational behavior represents intra-kernel dependencies, where the same kernel running on all devices requires reading or writing to a memory space it does not manage.

With inter-kernel dependencies, we define dependencies which result from data sharing between two different kernels of the same application. In this case, a succeeding kernel may have a data access map which is different from the data access map of the first kernel. For instance, kernel k_2 which is executed immediately after kernel k_1 may require access to some data produced by kernel k_1, but since the kernel k_2 has a different data access map, required data are not readily available in the device memory. This behavior represents inter-kernel dependencies and can be exhibited by multi-kernel applications.

Additionally, data-parallel applications may include an outer iterative loop wrapped around the kernels such as HPC applications for iterative solvers and data access patterns may change at each iteration.

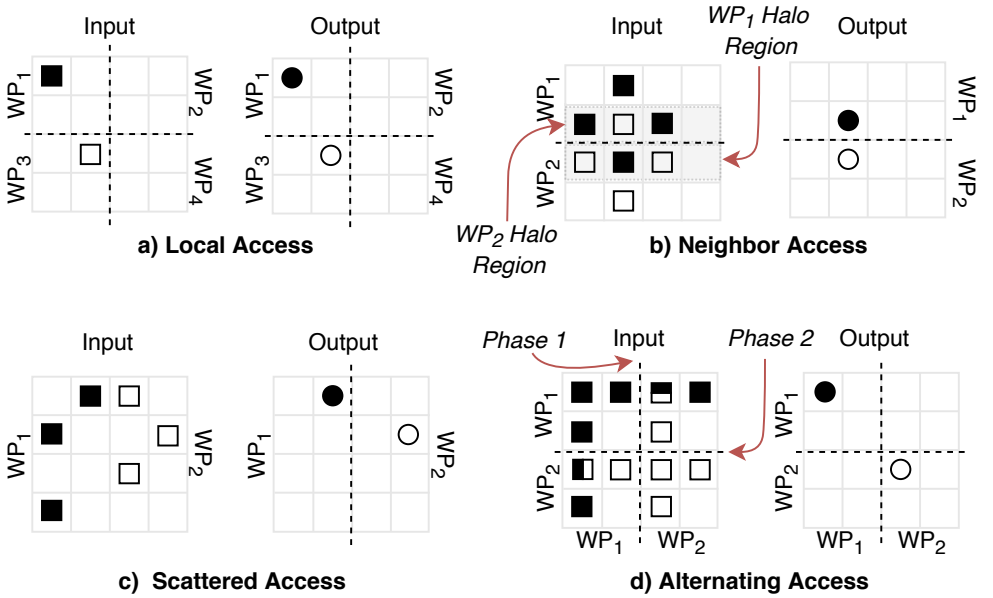


Figure 5.2.: Data Access Patterns

5.1.1. Intra-kernel Dependencies

Kernels of various HPC applications present with different data access patterns, which in case of partitioning of data may result in data dependencies, a critical factor for performance. Figure 5.2 shows various cases of data access patterns that data-parallel kernels exhibit. Two-dimensional 4x4 matrices present arrays of input and output. With squares we denote input data, while with circles output data generated by the execution process. For each of the cases, we have shown two different computations which generate two output data points. Black figures represent elements accessed by one work-item, while white figures represent elements accessed by another work-item.

Considering our work package approach communication between devices is not only expensive in terms of performance, but also very difficult to manage. Our goal is to analyze data access requirements of HPC kernels after the partitioning process, in order to be able to devise workload and kernel partitioning strategies which would result in no or minimal communication.

Local Access. Some types of data-parallel applications have a high degree of parallelism. As shown in Figure 5.2a, each element of the output array is generated by an element of the input array with the same index. We have used 2D-block partitioning, creating four work packages out of input and output domain, which can be scheduled independently. This case can be extended further such as if the whole row of a two-dimensional array is used to produce an element of the output array. For such applications, workload partitioning is simple and data to be accessed by the kernel code are always packed in the

local work package which is available in the device memory.

Neighbor Access. Some kernel codes, after data partitioning require access to neighboring elements which are packed in other work packages regardless of the spatial partitioning method used. Most commonly, these applications are scientific simulations such as heat propagation. In Figure 5.2b we have shown a case where neighboring elements are used for calculation of each output element. In such cases, there does not exist a partitioning method for which the kernel code could be transformed to access only local work package data. However, as depicted with a gray box in the figure, the so-called *halo regions* can be used in these cases. For each work package created by the partitioning process, the neighboring elements of the work package which need to be accessed by the kernel code are packed together with the work package and deployed to the devices. All output work package elements can be calculated without data transfers between devices, although, a minor overhead for transferring halo regions is to be expected. In most cases, halo regions account only for a very small fraction of data compared to the amount of data packed in work packages.

Scattered Access. In some cases kernel code has a very scattered data access map. Kernel code, as shown in Figure 5.2c requires access to distant elements which cannot be covered by halo regions. A stepwise approach to this problem is required. First, based on data access patterns, we look for a spatial partitioning method which can transform the access requirements to local work package access or work packages with halo regions. In some cases such as in Figure 5.2c, data accesses are scattered and irregular but confined to a region of the array. Black squares are located in the left part of the two-dimensional array while white squares in the right part. However, if data are scattered in such a way that no partitioning method results in local access, we next analyze potential partitioning strategies - which of the data domains to be partitioned makes more sense. For instance, if the input array (see figure 5.2) can fit into the memory of the devices, the input array can be replicated to all devices, while the output array is partitioned into work packages. In this case, for each output work package, we will have all input data available in all devices, while the workload has been split through output partitioning. Finally, in rare cases where both of the previous approaches do not overcome the problem, the computation is partitioned. Input and output are partitioned and for each output work package, the kernel is executed over all of the input work packages that contain the required data. At each execution stage (i.e. for each input work package) partial output elements are created which are aggregated at the end of the process. The last approach is used only as a last resort solution, where the input data are too big to fit in the device memory and the only way to execute the kernel is by data partitioning.

Alternating Access. Some kernels sweep over different regions of the array in phases. For instance, in Figure 5.2d the kernel that produces an element of the output array sweeps over all elements of one row, to switch later to sweeping

through all the elements of one column. In these cases splitting the kernel code into two or more kernels which run in sequence (kernel fission) can transform the access requirements to local work package access. In the case shown in the figure, in the first phase work packages are created based on a row-wise partitioning of data. A synchronization barrier is inserted after the execution of the first kernel, resulting in merging of the work packages in front-end. New work packages are created based on a column-wise partitioning and dispatched to devices where the output work packages are updated.

5.1.2. Inter-kernel Dependencies

Multi Kernel HPC Applications are data-parallel applications consisting of several kernels which have to be executed across a limited number of big datasets. Adapting clusterCL to support multi-kernel applications requires implementing features which handle together workload partitioning and the kernel execution schedule. In this section, we focus on inter-kernel dependencies where kernels of an HPC application executing in a serial sequence share data between each other.

Inter-kernel dependencies can be handled more simply than intra-kernel dependencies. In the context of intra-kernel dependencies, accessing data in other work packages occur during the execution process of the kernel. Data transfers to support kernel access patterns in multi-kernel execution occur at well-defined points between explicit calls for kernel execution and can be managed by the CPU with synchronization mechanisms or work package rerouting.

We will refer to Figure 5.1 to illustrate inter-kernel dependencies in the context of multi-kernel execution. Lets assume first that kernel k_1 and k_2 do not share data between each other, i.e. no inter-kernel dependencies. Assuming no intra-kernel dependencies exist the execution process is translated into a sequence of kernels each accessing its own local data in the devices where they are being executed. In this case, kernel k_1 needs a to produce $b_i, 0 < i < n$, therefore a is first transferred, while kernel k_2 needs c and $d_i, 0 < i < n$ to produce $e_i, 0 < i < n$. c is transferred in the beginning while work packages d_i are dispatched to devices before the commencement of execution for k_2.

In most cases, the succeeding kernel requires the output of the preceding kernel to produce its results. In cases where kernels share data between them, the sequence of the execution is not disrupted as long as the data access maps of the kernels match. For instance if kernel k_2 uses output array b_i to update values of d_i, such that each element of d_i is updated by the corresponding element of the b_i, the update process is confined to local access and we get $d_1 \leftarrow b_1, \cdots, d_n \leftarrow b_n$. Since work package b_i is already in the device memory as a result of the execution of the kernel k_1, k_2 can access them directly to update work packages d_i. In other cases, a synchronization barrier (inter-kernel synchronization) is imminent.

Lets assume that output array b from kernel k_1 is used as input c for kernel

k_2, such that $c \leftarrow b$. According to the partitioning strategy we have used in the case depicted in the Figure 5.1, c is a non-partitioned input array, therefore, work packages of b can not replace c for kernel k_2, since work packages of b are spread over all devices, while c is replicated to all devices. If no other partitioning strategy is possible, a synchronization barrier between kernels is required to ensure merging of work packages $b_1, \cdots, b_n$ into b at front-end, update and re-dispatch of c to devices in order to provide data coherence before the commencement of execution of kernel k_2.

Another case is when kernel k_2 requires access to b but has another map of data access. For instance, data access pattern shifts in the succeeding kernel such that to update work package d_i the kernel requires access to data packed in work package b_{i-1}. In these cases, front-end may reroute processed work packages b_i to other devices and nodes as needed, instead of using the synchronization mechanism which requires round-trips of work packages to and from the front-end. A more challenging case is if the data access map changes in such a way that rerouting is not useful. In these cases a synchronization barrier is inserted between kernels, data are merged in the front-end, repartitioned accordingly with data access requirements of the succeeding kernel and dispatched to devices.

Although a synchronization barrier appears like a diminishing factor in terms of performance, using the work package approach, its impact is highly reduced. The transfer of work packages is realized while the devices are executing other work packages. Therefore devices are not idle during the transfer process. An exception is the transfer of last work packages to the front-end and the first work packages to the nodes after synchronization for cases which require all WPs to be present at the front-end before the commencement of merging or repartitioning. Even in these cases, the transfer costs are mostly negligible, especially when the execution time dominates data transfer time.

A special case of HPC applications that exhibit similar dependencies as multi-kernel applications with inter-kernel dependencies are applications with iterative kernels. They usually consist of one or more kernels which have to be computed iteratively until some condition is true. For these applications, dependencies may occur at the iteration step. Hence, a synchronization barrier at the iteration step is required where merging and repartitioning of data is done.

5.2. Overlapping Data Transfers with Execution

Impact of the data transfer time on the overall execution time can be significant, especially for data-intensive applications. Considering that our model targets cluster systems, the process of preparing WPs from the process of execution in the devices are separated by two links: transfer from front-end to nodes and transfer from node DRAM to the device memory. The former is much slower than the latter and the main contributor to overall performance degradation.

We address this challenge with a prefetcher in order to minimize the effect of data transfer over Fast Ethernet/Infiniband. The basic idea is to override the dynamic on-demand fetching of WPs from the front-end by storing additional WPs on the node to keep devices - ideally - always WP-busy. The prefetcher fetches additional WPs while devices on the nodes are executing. However, the number of stockpiled WPs has to be carefully calculated, since storing a larger number of WPs might result in workload imbalance between cluster nodes.

With the device profiling information provided from the WP payload tuning process and considering the maximal throughput of the link between front-end and cluster nodes, the prefetcher has been upgraded to include calculation of several factors such as the trade-off between device execution time and data transfer time, number of nodes used in the computing, speed of all links and the potential latency of front-end arising from serving several nodes.

We evaluate the efficiency of our prefetcher feature by measuring the device WP-busy time and device idle time in Section 5.4.3.

5.3. Partitioning Strategies for HPC Applications

We have analyzed and tested 11 HPC applications from different benchmarks and application suites: AMD Accelerated Parallel Processing SDK [AMD], NVIDIA OpenCL SDK [nVi19b] and PolyBench [GXS+12]. Benchmark applications are summarized in Table 5.1. We have adapted these applications and their kernels to allow for multi-device computation.

Table 5.1.: List of HPC applications ported to clusterCL

Application	**Code**	**Domain**
Direct Coulomb Summation	DCS	Scientific simulation
Correlation Matrix	COR	Data Mining
Covariance Matrix	COV	Data Mining
Binomial Options Pricing Model	BOP	Finance
Matrix-Matrix Multiplication	GEMM	Linear Algebra
Gram-Schmidt Process	GSP	Linear Algebra
Rank-K Matrix Operations	SYRK	Linear Algebra
Black-Scholes Model	BSM	Finance
Convolution	CONV	Scientific simulation
Biconjugate Gradient	BICG	Linear Algebra
Scalar, Vector and Matrix Multiplication	GSMV	Linear Algebra

Following we discuss in more detail the partitioning strategies and kernel scheduling techniques for the listed applications in Table 5.1.

Direct Coloumb Summation. DCS is used to calculate the electrostatic field for a given n-dimensional grid space based on the charges of the atoms

within that grid space. It consists of an input array which stores atom characteristics and an output array which stores data about the energy grid. For this application, either the input or the output domain can be partitioned while the mixed partitioning is impractical. Whichever domain is partitioned the other must be replicated to all devices.

Correlation Matrix and Covariance Matrix. COR and COV compute the correlation, respectively covariance matrix. Since the first three kernels are lightweight, they can be executed by all devices, while the fourth kernel requires output domain partitioning. Output domain is a triangular matrix making column-wise partitioning and work package payload size tuning highly advisable for better performance.

Binomial Options Pricing Model. BOP calculates binomial options pricing for European Options. Output domain partitioning is the most beneficial partitioning strategy.

Matrix-Matrix Multiplication. GEMM is a typical BLAS application. Partial input and output domain partitioning is the most beneficial partitioning strategy since a complete partitioning of both input and output introduces redundant iterations over work packages slowing down the execution.

Gram-Schmidt Process. GSP is a linear algebra application used for orthonormalization of a set of vectors in an inner product space. A complete partitioning strategy is possible. However, an inter-kernel synchronization barrier is required before the execution of the third kernel. A time-step loop over the multi-kernel sequence is necessary, although it does not affect the synchronization or the data transfer process.

Rank-K Matrix Operations. SYRK is a linear algebra application used for symmetric rank-k update operations. A partitioning strategy similar to GEMM is appropriate.

Black-Scholes Model. BSM is an implementation of option pricing model for European Options in financial engineering. A complete partitioning of input and output arrays is appropriate.

Convolution. CONV is an application used in the field of neural networks. Our implementation is based on a 2D version of the application. A complete partitioning of input and output arrays is appropriate.

BiConjugate Gradient Method. BICG is a linear algebra application. A complete partitioning of input and output arrays is appropriate.

Scalar, Vector and Matrix Multiplication. GSMV is a linear algebra application. Partial input domain and complete output domain partitioning are possible.

5.4. Experimental Evaluation

In this section we present and discuss the results of the evaluation of *clusterCL* framework with our benchmark suite (see Table 5.1) and three clusters.

We focus our evaluation in the following aspects of the clusterCL framework that show:

- The applicability of our approach, which is examined in section 5.4.2 where the speedup figures for symmetric clusters are analyzed.
- Workload balance among cluster nodes, including the impact of data transfers in heterogeneous asymmetric configurations is examined in section 5.4.3
- The benefits of partition tuning for WPs are shown in section 5.4.4

5.4.1. Experimental setup

Hardware Setup

In our experiments, we use 3 clusters: PHIA, EXA and CORA. Specifications of the cluster systems used to evaluate *clusterCL* are summarized in Table 5.2. All clusters are heterogeneous. Clusters PHIA and EXA have an asymmetric configuration with cluster nodes hosting different types of compute devices, while CORA is symmetric.

Table 5.2.: Cluster Systems

Cluster	# of nodes	Interconnection	Nodes	Configuration
PHIA	8	InfiniBand QDR (40 Gbps)	1 - 6	CPU: 2 x Intel Xeon E5-2650 GPU: 2 x NVIDIA Tesla K20m OTH: 2 x Intel Xeon Phi 5110P
			7	CPU: 2 x Intel Xeon E5-2650 GPU: 4 x NVIDIA Tesla K20m
			8	CPU: 2 x Intel Xeon E5-2650 OTH: 4 x Intel Xeon Phi 5110P
EXA	4	InfiniBand FDR (56 Gbps)	1	CPU: 4 x Intel Xeon Gold 6138
			2	CPU: 2 x AMD EPYC 7501
			3	CPU: 2 x Intel Xeon Gold 6130 GPU: 1 x Nvidia Tesla V100
			4	CPU: 2 x Intel Xeon Gold 6130 GPU: 1 x AMD Rad. Inst. MI25
CORA	7	Ethernet (1 Gbps)	1 - 7	CPU: Intel Xeon X5550 GPU: 2 x NVIDIA Tesla C2050 1 x NVIDIA Tesla C1060

PHIA is interconnected with Infiniband QDR (40 Gbps) and consists of 8 nodes. First six nodes have an identical configuration hosting NVIDIA K20m and Xeon Phi along with the Intel CPUs. Node 7 and 8 host K20ms or Xeon Phi only. EXA is interconnected with InfiniBand FDR (QSFP, 56 Gbps) and consists of 4 compute nodes with Intel and AMD CPUs, NVIDIA Tesla V100 and AMD Radeon Instict MI25 GPUs. CORA is interconnected with Fast Ethernet and consists of 7 nodes which host NVIDIA Tesla C2050 and Tesla C1060 GPUs.

We use OpenMPI 4.0.1 in EXA and CORA, and MVAPICH 2.2 in PHIA for inter-node communications. For NVIDIA devices, we use NVIDIA's OpenCL implementation, for AMD devices - rOCM platform, and for Intel devices Intel's implementation of OpenCL.

Cluster configurations

We have used different cluster configurations to examine the behavior of the clusterCL framework for our evaluation objectives. As shown in Table 5.3, PSC1 - PSC6 are six symmetrical configurations of PHIA cluster consisting of 1 to 6 nodes respectively, where we use all compute devices (CPUs, K20m, Xeon Phi) to run experiments.

Other configurations are asymmetric, with PAC1, PAC2 and PAC3 consisting of nodes 6, 7 and 8 of PHIA cluster. In PAC1 we have used GPUs and Xeon Phi which are distributed unevenly among PHIA nodes 6, 7 and 8. In PAC2 we use only GPUs, therefore node 8 is not used since it does not contain any GPU. In PAC3 we use only Xeon Phi, therefore node 7 is discarded since no Xeon Phi are installed in that node. EAC1 is an asymmetric configuration of EXA cluster consisting of nodes 3 and 4 with their respective compute devices, NVIDIA Tesla V100 in node 3 and AMD Radeon Instinct MI25 in node 4. In CAC1 we have constructed an artificial asymmetric configuration by utilizing all three of available GPUs (two Tesla C2050 and one Tesla C1060) in the first node, followed by two devices in the second node (only Tesla C2050s) and one device in the third node (only Tesla C1060).

Applications with Different Problem Sizes.

We have experimented with applications from various benchmarks as detailed in section 5.3. For the first two experiments we have used six different problem sizes (PS1 - PS6), with problem sizes in the lower end (PS1 - PS3) that can fit into the smallest device memory and problem sizes in the higher end (PS4 - PS6) that cannot be computed without partitioning the data domain between devices.

Table 5.3.: Experimental configurations

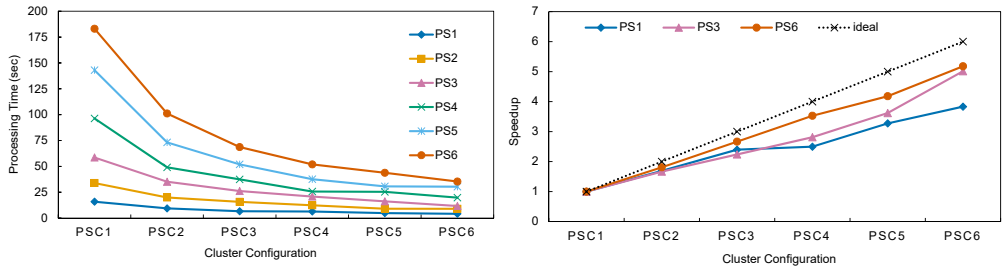
Configuration	Cluster	Arrangement	Details
PSC1	PHIA	Symmetric	PSC1: Node 1 (all devices)
⋮	⋮	⋮	⋮
PSC6	PHIA	Symmetric	PSC6: Nodes 1 - 6 (all devices)
PAC1	PHIA	Asymmetric	Nodes 6 - 8 (GPUs & Xeon Phi)
PAC2	PHIA	Asymmetric	Nodes 6 - 8 (only GPUs)
PAC3	PHIA	Asymmetric	Nodes 6 - 8 (only Xeon Phi)
EAC1	EXA	Asymmetric	Nodes 3 & 4 (GPUs)
CAC1	CORA	Asymmetric	Node 1 (all GPUs), Node 2 (only Tesla C2050s), Node 3 (only Tesla C1060)

5.4.2. clusterCL Speedup Analysis

In this experiment, we compare the performance of clusterCL, the processing times and the speedup for some of the applications with different problem sizes and cluster configurations. Since the application workload is partitioned into many work packages, we use *processing time* to denote the maximal accumulated handling time for all work packages assigned to the devices of the system. Handling time includes writing data from the node to the device, execution time in the device and time required to read the results from the device to the cluster node.

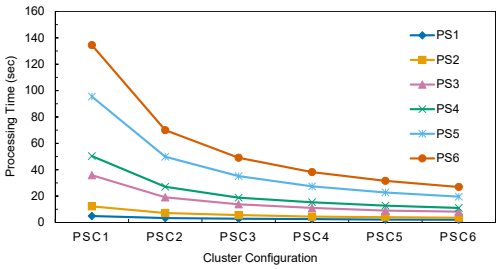
Experiments are carried out in our PHIA cluster for cluster configurations PSC1 - PSC6. For the speedup, we use a single node configuration (PSC1) as a baseline to test the flexibility of the framework in work distribution for several compute nodes with identical configuration (PSC2 - PSC6, see Table 5.3). A dotted line is shown in all figures to denote the ideal speedup figures (theoretical peaks for each configuration).

In Figure 5.3a we show the processing times for BOP. Different problem sizes (PS1 - PS6) are plotted using different curves, from the smallest problem size processing times (the bottommost curve) to the biggest problem size (the uppermost curve). Processing times (vertical axis) are given for each of the cluster configurations (PSC1 - PSC6) (horizontal axis). Figures on the right such as Figure 5.3b show the relative speedup compared to baseline for problem sizes (PS1, PS3 and PS6) - we have omitted problem sizes PS2, PS4, PS5 for the sake of readability. For most of the applications shown in Figure 5.3, respectively figures 5.3a - 5.3j the speedup figures approach the ideal speedup, especially with bigger problem sizes which show the strongest performance gain. BOP speeds up to 5.17x over the baseline version, GEMM, COV, DCS and

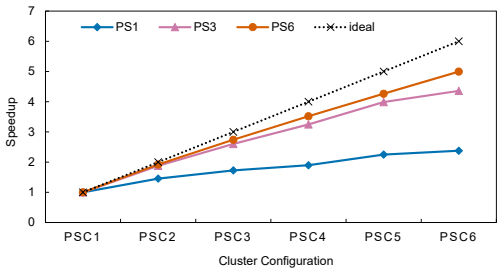


(a) BOP: Processing Time

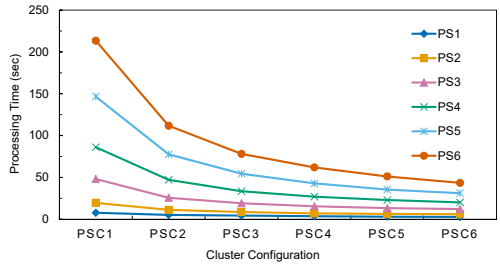
(b) BOP: Speedup



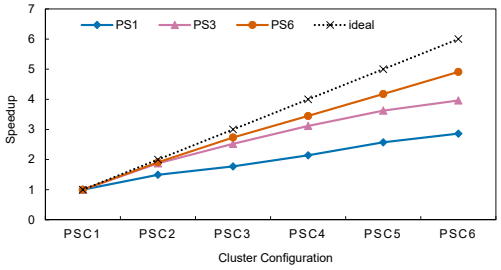
(c) GEMM: Processing Time



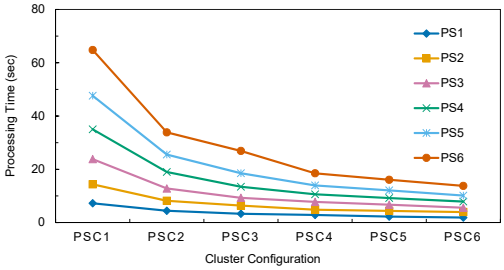
(d) GEMM: Speedup



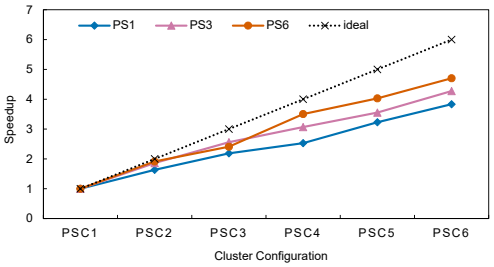
(e) COV: Processing Time



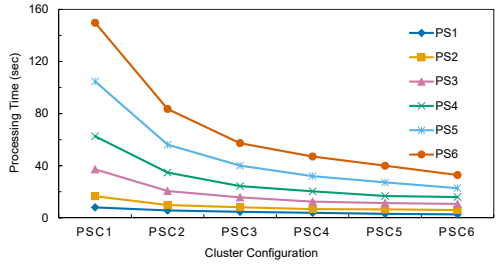
(f) COV: Speedup



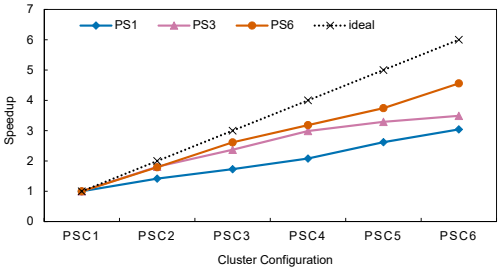
(g) DCS: Processing Time



(h) DCS: Speedup

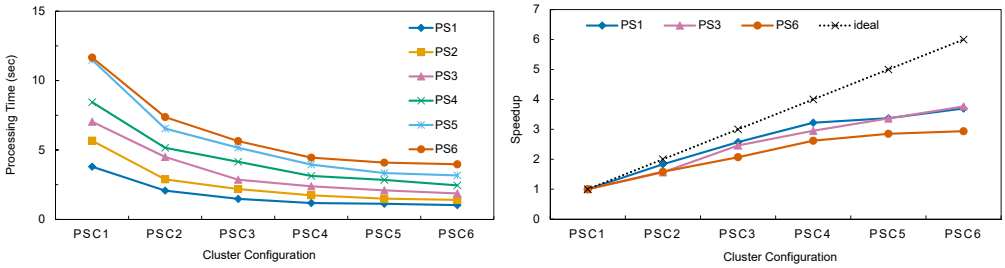


(i) COR: Processing Time



(j) COR: Speedup

Figure 5.3.: clusterCL: Processing Time and Speedup for HPC Applications in Symmetric Heterogeneous Clusters with different cluster configurations



(k) BICG: Processing Time

(l) BICG: Speedup

Figure 5.3.: clusterCL: Processing Time & Speedup for HPC Applications in Symmetric Heterogeneous Clusters with different cluster configurations

COR up to 4.99x, 4.91x, 4.7x and 4.55x respectively in a PSC6 configuration for biggest problem size PS6. The gap between the ideal speedup and real speedup figures is mainly due to inter-node data transfers. While for single-node execution there are no WP transfers to other nodes, in other configurations transfer of the WPs cannot be hidden by the prefetcher until the execution commences in the devices. Upon the start of the computing process in the devices, the prefetcher transfers other work packages while the devices are busy executing, effectively minimizing the packet switching time. We observe from the speedup figures that this gap is bigger for small problem sizes and smaller for bigger problem sizes. As the problem size increases, so does the number of work packages created by the partitioning process and the overall execution time. Since we have the same configurations and the same number of devices, data transfer time for the first work packages which cannot be hidden by the prefetcher stays the same. Therefore, the impact of the time required to transfer first work packages in overall execution time is reduced proportionally with the increase in the number of work packages. It is worth noting that in all of these experiments we can observe that the speedup curves for bigger problem sizes are projected in such a way that strong performance is expected to be sustained even for system configurations with more identical nodes and bigger problem sizes.

In Figure 5.3d and Figure 5.3j for GEMM and COR for problem size PS3 it appears like speedup figures diminish as the scale of the system gets bigger. Our observations show that this is due to scheduling. For this experiment, we are using a work-stealing strategy where the device which finishes first gets the next work package. Big differences in execution times among compute devices (e.g. CPU executes a WP seven times slower than the GPU for GEMM) result in workload imbalance due to the small number of work packages. In both cases, the CPU is getting one of the last WPs and while other devices complete execution, the whole process is delayed waiting for the CPU to finish processing.

This can be seen in the respective figures which show the speedups for GEMM for configuration PSC6, and for COR for configurations PSC5 and PSC6. This problem can be solved by applying a static scheduling algorithm such as the one proposed in our paper that deals with scheduling problems in heterogeneous clusters [RMH16].

In Figure 5.3k and 5.3l we have shown processing time and speedup for BICG over the baseline. As shown in the figure, the speedup over the baseline is around 3x. These figures also resemble the results for GSMV, BSM, CONV and SYRK applications which we omitted for readability reasons. In Figure 5.3k we see that for system configuration PSC6 even for the bigger problem sizes the overall execution time is under 5 seconds. These applications fall into the category of data-intensive applications with low computing intensity. A big ratio of data transfer time over execution time hampers the distribution process and inhibits the prefetcher to supply enough work packages in time. In those cases using of work package payload size tuner in conjunction with increasing thread granularity can improve performance.

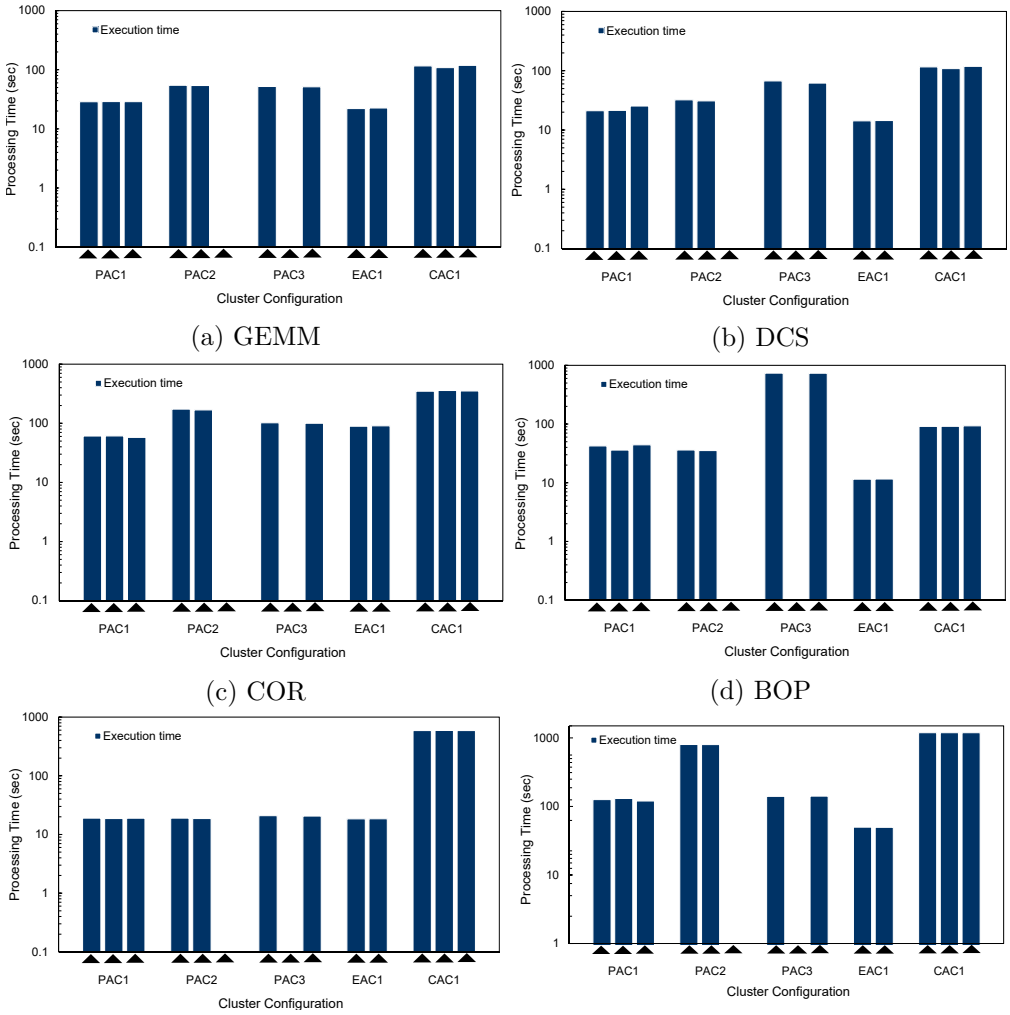
Finally, our observations show that achieving sustained performance in heterogeneous clusters requires adaptation of multiple factors such as workload partitioning decisions and scheduling decisions with the aim of decreasing the impact of data transfers in the computing process.

5.4.3. Workload Balance, Prefetcher Efficiency and Impact of Synchronizations

In this section, we examine the efficacy of our approach in guaranteeing workload balance between cluster nodes for asymmetric configurations of heterogeneous clusters. Additionally, we show the work share between different types of devices and discuss the efficiency of our prefetcher module in minimizing packet switching time during the computing process. Impact of synchronizations in the execution process is also investigated in this section. For this experiment, we use asymmetric configurations of our three clusters as detailed in Table 5.3.

Figure 5.4 shows the workload balance between cluster nodes different applications such as GEMM, DCS, COR, BOP, BSM and GSP. Execution time (vertical axis) is set in the logarithmic scale — triangles in the horizontal axis act as placeholders for the nodes of a specific configuration. In Figure 5.4 for PAC1 cluster configuration execution times are shown for each of the three nodes, while execution times are shown only for two nodes for configurations PAC2 and PAC3. In the case of PAC2, no GPUs are installed on node 8, so execution takes place only in nodes 6 and 7. In a similar fashion in PAC3 no Xeon Phi is installed in node 7, therefore execution takes place only in nodes 6 and 8. For EAC1 we show execution times for nodes 3 and 4 (nodes consisting of heterogeneous devices), and for CORA we show execution times for nodes 1, 2 and 3.

Additionally, in Tables 5.4 - 5.9 we show the experimental data for the same



(a) GEMM (b) DCS

(c) COR (d) BOP

(e) BSM (f) GSP

Figure 5.4.: clusterCL: Workload Balance between cluster nodes in various Asymmetric Cluster Configurations

group of applications. The first column-group shows the work share between devices of different types for each configuration. The amount of each type of devices in a configuration are specified in Table 5.3. The second column-group shows average WP-busy time per device, which means the time devices spent on processing work packages. The remainder, in this case, is device idle time. Third column-group shows average execution time per work package for a specific device type.

In Figure 5.4a we see the workload balance for GEMM in different cluster configurations. For PAC1, we see that the relative standard deviation from the mean (%rsd) of processing times is less than 3%, receptively all nodes finish

processing within ±1.5 seconds, while the overall execution time of the application is 32.89 seconds. For PAC2, PAC3, EAC1 and CAC1 configurations $\%rsd$ is even lower at 0.58%, 0.98%, 1.33% and 2.04% respectively. In experiments with other applications (see Figures 5.4b - 5.4f), we observe that relative standard deviation from the mean in processing times between nodes is in the range of $0.07\% < \%rsd < 1.5\%$, which as in the case of GEMM show a good balance of workload among computing nodes. We also see few exceptions where $1.5\% < \%rsd < 5\%$ which are usually associated with PAC1 configuration such as for DCS, COR, BOP and GSP. According to our experimental data, discrepancies in runtime between nodes in these cases have all come as a result of work-stealing scheduling decisions, similar to the problem we have described in the previous section for GEMM and COR for problem size PS3. This problem can be fixed by using our best performance algorithm for work distribution, which we will discuss in detail in Part III.

Another interesting observation is the discrepancies in processing time for some types of applications in different types of devices. For PAC2 configuration we use only Tesla K20m GPUs and for PAC3 we use only Xeon Phi. A close observation of Figure 5.4d and 5.4f shows that some types of applications such as BOP fit much better for a PAC2 configuration, respectively for a GPU architecture, while we observe the opposite for GSP in PAC3 configuration, which seems to fit better with a Xeon Phi architecture. This is supported by experimental data in Tables 5.7 and 5.9 where the processing times per WP are notably different between K20m and Xeon Phi. In a combined approach with GPUs and Xeon Phi such as the PAC1 configuration, these substantial differences in execution time do not degrade overall application performance since a bigger amount of work packages shifts towards faster devices ensuring workload balance and faster completion of execution. We observe this in case of BOP where the GPUs have a bigger share of work packages (see Table 5.7), while in the case of GSP the share of work packages is more balanced, tilting more towards Xeon Phi (see Table 5.9).

Next, we investigate the efficiency of the prefetcher module. As shown in Tables 5.4 - 5.9 the second column-group - average WP-busy time per device - shows the device utilization during the computing process, i.e. the ratio of the overall execution time during which the devices are not idle. We see that for all of the applications, except for BSM, the utilization of the devices is more than 98% (in most cases more than 99.5%), which translates that the device idle time, or packet switching time by the prefetcher is less than 2% of the overall computing time for an application. This shows strong efficiency of clusterCL in handling heterogeneous asymmetric cluster by overlapping data transfer with execution and amortization of the impact of data transfers in computing.

On the other hand, for BSM (see Table 5.8) device idle time is relatively high. BSM is a data-intensive application with very low arithmetic intensity and data transfer times are orders of magnitudes higher than kernel execution times. The

Table 5.4.: GEMM: Work share between devices

	Work share (%)		Avg. WP-busy time per device (%)		Avg. proc. time per WP (sec)	
	Tesla K20m	**Xeon Phi**	**Tesla K20m**	**Xeon Phi**	**Tesla K20m**	**Xeon Phi**
PAC1	51.12%	48.88%	99.90%	99.70%	0.70	0.59
PAC2	100.00%	-	99.94%	-	0.70	-
PAC3	-	100.00%	-	99.70%	-	0.59
	Tesla V100	**AMD MI25**	**Tesla V100**	**AMD MI25**	**Tesla V100**	**AMD MI25**
EAC1	77.46%	22.54%	100.00%	100.00%	0.06	0.21
	Tesla C2050	**Tesla C1060**	**Tesla C2050**	**Tesla C1060**	**Tesla C2050**	**Tesla C1060**
CAC1	83.04%	16.96%	93.90%	96.89%	0.55	1.45

Table 5.5.: DCS: Work share between devices

	Work share (%)		Avg. WP-busy time per device (%)		Avg. proc. time per WP (sec)	
	Tesla K20m	**Xeon Phi**	**Tesla K20m**	**Xeon Phi**	**Tesla K20m**	**Xeon Phi**
PAC1	66.96%	33.04%	98.95%	99.07%	1.59	3.23
PAC2	100.00%	-	99.68%	-	1.59	-
PAC3	-	100.00%	-	99.74%	-	3.23
	Tesla V100	**AMD MI25**	**Tesla V100**	**AMD MI25**	**Tesla V100**	**AMD MI25**
EAC1	62.50%	37.50%	99.90%	99.97%	0.05	0.08
	Tesla C2050	**Tesla C1060**	**Tesla C2050**	**Tesla C1060**	**Tesla C2050**	**Tesla C1060**
CAC1	89.29%	10.71%	99.63%	99.00%	2.16	9.53

Table 5.6.: COR: Work share between devices

	Work share (%)		Avg. WP-busy time per device (%)		Avg. proc. time per WP (sec)	
	Tesla K20m	**Xeon Phi**	**Tesla K20m**	**Xeon Phi**	**Tesla K20m**	**Xeon Phi**
PAC1	49.78%	50.22%	99.81%	99.75%	1.53	1.26
PAC2	100.00%	-	99.93%	-	2.23	-
PAC3	-	100.00%	-	99.90%	-	1.24
	Tesla V100	**AMD MI25**	**Tesla V100**	**AMD MI25**	**Tesla V100**	**AMD MI25**
EAC1	68.97%	31.03%	99.83%	100.00%	0.28	0.64
	Tesla C2050	**Tesla C1060**	**Tesla C2050**	**Tesla C1060**	**Tesla C2050**	**Tesla C1060**
CAC1	64.51%	35.49%	99.77%	99.85%	4.81	4.39

slowest gear in cluster configurations is the transmission of data over Infiniband or in case of CORA over Fast Ethernet. When a massive amount of data has to be transferred to the nodes and computing intensity of the application is low (e.g. data-intensive applications), the prefetcher is incapable of overlapping data transfers with execution, i.e. eliminate packet switching time. In this case, the data transfer time for one WP from front-end to the node was 60 times more than the execution of that WP in K20m, respectively 12 times more than the execution in Xeon Phi. However, as long as the problem size of the application is bigger than the capacity of one device or the collective

Table 5.7.: BOP: Work share between devices

	Work share (%)		Avg. WP-busy time per device (%)		Avg. proc. time per WP (sec)	
	Tesla K20m	**Xeon Phi**	**Tesla K20m**	**Xeon Phi**	**Tesla K20m**	**Xeon Phi**
PAC1	94.64%	5.36%	91.32%	99.96%	0.46	9.77
PAC2	100.00%	-	99.70%	-	0.46	-
PAC3	-	100.00%	-	99.98%	-	9.75
	Tesla V100	**AMD MI25**	**Tesla V100**	**AMD MI25**	**Tesla V100**	**AMD MI25**
EAC1	67.86%	32.14%	99.91%	99.99%	0.03	0.08
	Tesla C2050	**Tesla C1060**	**Tesla C2050**	**Tesla C1060**	**Tesla C2050**	**Tesla C1060**
CAC1	81.47%	18.53%	97.97%	97.55%	0.95	2.14

Table 5.8.: BSM: Work share between devices

	Work share (%)		Avg. WP-busy time per device (%)		Avg. proc. time per WP (sec)	
	Tesla K20m	**Xeon Phi**	**Tesla K20m**	**Xeon Phi**	**Tesla K20m**	**Xeon Phi**
PAC1	50.67%	49.33%	17.19%	11.81%	4.E-04	2.E-03
PAC2	100.00%	-	28.07%	-	4.E-04	-
PAC3	-	100.00%	-	32.57%	-	2.E-03
	Tesla V100	**AMD MI25**	**Tesla V100**	**AMD MI25**	**Tesla V100**	**AMD MI25**
EAC1	67.75%	32.25%	48.57%	32.89%	7.E-05	8.E-04
	Tesla C2050	**Tesla C1060**	**Tesla C2050**	**Tesla C1060**	**Tesla C2050**	**Tesla C1060**
CAC1	56.14%	43.86%	1.69%	2.43%	1.E-03	6.E-03

Table 5.9.: GSP: Work share between devices

	Work share (%)		Avg. WP-busy time per device (%)		Proc. time (range) per WP (sec)	
	Tesla K20m	**Xeon Phi**	**Tesla K20m**	**Xeon Phi**	**Tesla K20m**	**Xeon Phi**
PAC1	53.06%	46.94%	98.66%	99.2%	1.0E-05 - 27.3	2.0E-05 - 8.57
PAC2	100.00%	-	99.12%	-	1.0E-05 - 31.1	-
PAC3	-	100.00%	-	98.94%	-	2.E-03 - 10.1
	Tesla V100	**AMD MI25**	**Tesla V100**	**AMD MI25**	**Tesla V100**	**AMD MI25**
EAC1	81.05%	18.95%	99.3%	99.15%	4.0E-06 - 1.32	2.0E-03 - 11.13
	Tesla C2050	**Tesla C1060**	**Tesla C2050**	**Tesla C1060**	**Tesla C2050**	**Tesla C1060**
CAC1	64.71%	35.29%	94.52%	93.56%	3.0E-03 - 48.32	1.0E-02 - 112.34

capacities of all devices in one node, workload partitioning between the nodes is unavoidable and lower device utilization numbers are acceptable in case of data-intensive applications.

Finally, we examine the impact of synchronizations for applications with dependencies. GSP is a compute-intensive imbalanced application with iterative kernels, where the workload is determined by the iteration loop and spatial coordinates of array elements. In this case in Table 5.9, we have shown the range of processing times per work package in different devices instead of average processing times since different work packages have different workload due to application imbalance. Before the commencement of execution of the

third kernel, one of the arrays has to be merged and repartitioned by the front-end, since the third kernel has a different access map to that array compared to kernel 1 and 2. The overall execution times for GSP in PAC1, PAC2 and PAC3 configurations are 130.36s, 811.36s and 143.97s respectively. The time required for merging and repartitioning by the front-end is 0.45s±0.05s. Since most of the work packages are transferred while devices are busy, only last work packages sent to front-end before synchronization and first work packages dispatched to nodes after synchronizations account for the device idle time - data transfer time which cannot be overlapped with execution by the prefetcher. Average accumulated time for these transfers is 0.00144s±0.00002 depending on cluster configuration. The total device idle time due to synchronization is at most 0.46s in any of the configurations or 0.35%, 0.05%, 0.31% respectively for PAC1, PAC2 and PAC3 configurations. The impact of synchronization in overall execution time in this case is very small, mainly due to prefetching and high arithmetic intensity of the application. Experimental data for WP-busy time shown in Table 5.9 are slightly bigger than these numbers since they also account for packet switching time in addition to synchronization.

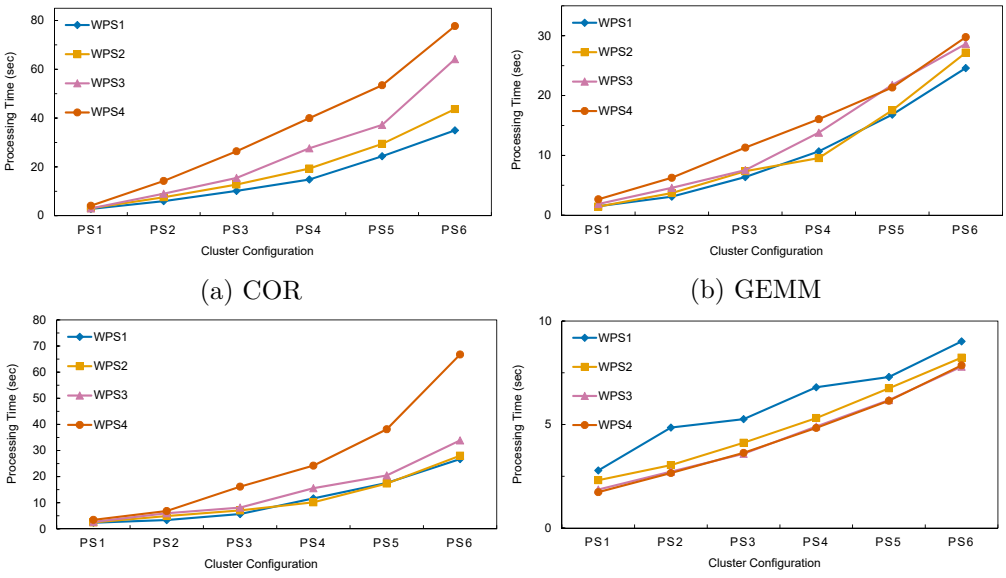
5.4.4. Partitioning Tuning

To test our approach for work package tuning as we have presented it in Section 3.1.3 we use different problem sizes (PS1 - PS6) for selected HPC applications COR, GEMM, GSP and CONV. Whereas in the experiments above we have used standard work package sizes (non-tuned), in this experiment we use the tuner to test a set of different work package sizes and show the execution times for work package sizes WPS1 - WPS4, where WPS1 is the smallest (usually 1/128 - 1/448 of the tested problem size for PS1 to PS6) and WPS4 the biggest work package size (usually 1/16 - 1/56 of the tested problem size for PS1 to PS6). We use the PSC6 symmetric cluster configuration for this experiment.

Figure 5.5 shows the execution times for different WP sizes. CORR and GSPM, as shown in the figure have the most remarkable performance improvements with speedups as much as 3x for the smallest tested WP size compared to the biggest WP size for PS6. This huge speedup can be attributed mainly to specific application characteristics. Both applications are imbalanced such that different work-items have different amount of work to process. Reducing the work package size balances better the workload among the work packages and the work-items have a more uniform distribution of computation load.

In the case of GEMM where the workload between the work-items is balanced, the differences in execution time are much smaller, however very important for large problem size applications as shown in Figure 5.5b — switching from the biggest tested WP size to the smallest results in a 21% performance improvement.

Finally, we can observe that for data-intensive applications such as CONV



(a) COR (b) GEMM

(c) GSP (d) CONV

Figure 5.5.: Effects of the WP Tuner in the overall execution time

(see Figure 5.5d), a bigger WP size is much more beneficial, since it increases the kernel execution times, which effectively enables the prefetcher module to better balance between execution time and data transfer time, respectively decreasing the packet switching time. As shown in the figure, choosing the biggest tested WP size over the smallest reduces the overall execution time by almost 15%.

Based on the results from the Figure 5.5 we can conclude that for all types of applications, compute-intensive, data-intensive and mostly for imbalanced application, use of the WP Tuner is beneficial and improves the overall performance in all cases.

5.5. Summary

Enabling the execution of multi-kernel data-parallel applications with dependencies is a crucial element of our framework. In this chapter, we put our effort in analyzing different application characteristics and patterns in order to group similar applications into categories for which strategies that apply to the whole group can be devised and implemented within the scope of clusterCL. We have implemented synchronization and barriers which help ensure data coherency over the whole system at well-defined points. We have analyzed the potential partitioning strategies for a broad group of HPC applications and evaluated the framework by showing the scalability, efficiency, workload balance, and perfor-

mance improvement from the tuning of work package payload sizes. Hence, we think our approach provides an effective and efficient way for executing HPC applications in heterogeneous cluster architectures.

Part III.

Optimal Work Distribution in Heterogeneous Clusters

CHAPTER 6

Efficiency-oriented Scheduling Strategies

The importance of heterogeneous asymmetric clusters has grown steadily over the last years. Such architectures pose new challenges with respect to execution time and energy consumption. Work distribution for homogeneous systems has been done typically by dividing the work by the number of compute nodes and assigning equal-sized portions to each of them. Such an approach is not adequate anymore for heterogeneous systems. Heterogeneity of devices raises in addition to execution time a second issue – the energy consumed to fulfill a task. A work distribution approach could, at its most basic function exclude low-performing, high-energy-consuming devices from execution. However, optimization in one direction only, i.e. execution time or energy consumption, does not meet the requirements. In this chapter, we discuss different work distribution approaches and lay the groundwork for optimizing executions based on multi-objective functions.

In heterogeneous clusters, energy-efficiency is a very important issue introduced by heterogeneous devices and it is gaining increasing attention. Efficiency-oriented scheduling strategies besides performance should take into account also energy consumption in a time-energy problem space.

In this and the following chapters, we consider applications with a workload that is partitioned into work packages which are assigned to devices to be processed. Runtime efficiency and energy efficiency basically boils down to the problem of mapping work packages to devices.

Different work distribution strategies are possible, resulting in different solutions exhibiting different efficiency characteristics. An assessment of a solution heavily depends on the requirements of the programmer. When we optimize in two directions, the mapping of work packages to devices results in a solution space containing 4 distinguished solutions exhibiting optimality criteria:

- *best performance*: one dimensional problem–energy consumption neglected
- *minimal energy*: one-dimensional problem–execution time neglected
- *minimal energy with restrictions to performance*: two-dimensional problem
- *best performance with restrictions to energy*: two dimensional problem

In addition to the 4 solutions above, trade-off solutions between time and energy can be considered as well. This chapter discusses all the different optimization problems and presents the impacts in practice.

6.1. Energy-efficient Work Distribution

The goal of the framework is to assist a programmer to run an application efficiently in a heterogeneous environment. Our strategy to deal with such clusters is to partition the work into work packages which are assigned to compute devices to be processed. As shown in Figure 6.1, the input is partitioned in smaller data chunks called *work packages* which are equal-sized.

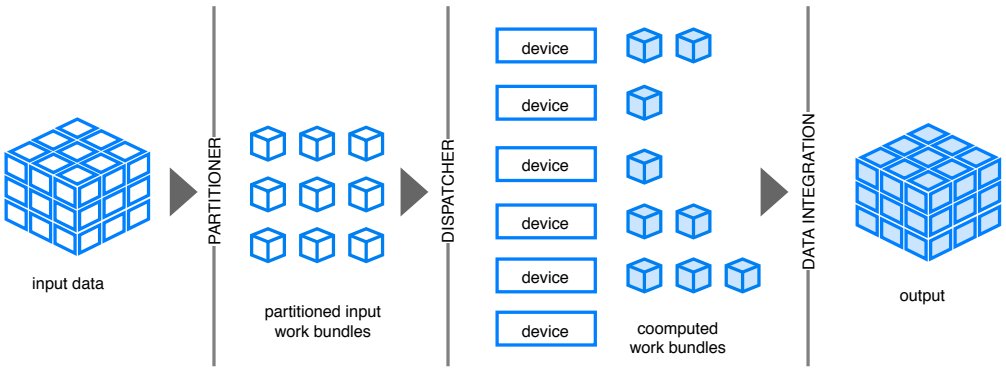


Figure 6.1.: Work distribution process

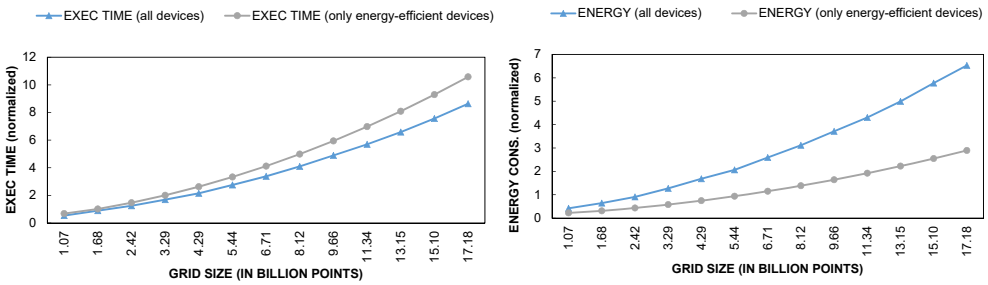
The size of work packages is determined by taking various hardware and application characteristics into account, like memory capacity or peak performance. Usually, the number of work packages are orders of magnitude higher than the number of available devices in the system. This problem, coupled with different performance and energy consumption numbers for each of the devices leads to the requirement for a strategy in distributing work packages.

Work distribution is done by mapping work packages onto compute devices. Hence, different mappings result in different execution times and different power draws. Depending on whether execution time or energy consumption is favored by a programmer, some mappings fit better to the needs than others. In the

following, we will discuss three examples which have the common property that better performance is achieved at the cost of higher energy consumption. Or in other words: longer execution times may help to save energy. These examples show that getting better in one dimension may degrade the other dimension, which means that optimization has to take both dimensions into account.

The first example is taken from a paper by Liu and Luk [LL12]. As shown there, the FPGA is the most energy-efficient device for executing function SGEMM (matrix-matrix operation $C = \alpha AB + \beta C$) followed by GPU and CPU. Using only the FPGA for processing SGEMM leads to the smallest amount of energy consumption, but it may take about 40 times longer to complete than using the GPU.

We have done a similar experiment on our PHIA cluster, which consists of 8 compute nodes with devices such as NVIDIA GPUs and Intel XEON Phi accelerators arranged in an asymmetric configuration. Our experimental results confirm that execution time and energy consumption can be influenced to a great extent based on the devices used in computing. In Figure 6.2, we show the execution time and energy consumption in normalized units for a molecular analysis kernel with different problem sizes. As it is shown in Figure 6.2a, when we use only most energy-efficient devices the execution time (circular markers) increases in comparison to the program version where we use all available devices of the system (triangular markers). The gap between the two execution scenarios increases steadily with bigger problem sizes. However, in Figure 6.2b it can be seen that energy consumption is reduced significantly when only the most energy-efficient devices are used in computation (circular markers). The figures show that for all problem sizes a loss in performance is rewarded by a reduction of energy consumption.



(a) Execution time for all devices used vs. energy-efficient devices only

(b) Energy consumption for all devices used vs. energy-efficient devices only

Figure 6.2.: Performance and energy figures in PHIA cluster for DCS

In addition to selecting devices for execution, the next example shows that the distribution of work onto the selected devices plays an important role as well. To demonstrate this, let us assume we have 5 equally-sized work packages and two devices with the following time and energy values per work package $T = \{2, 5\}$

and $E = \{10, 5\}$, i.e. 2 and 5 time units are taken for a work package on device 1 and 2, and 10 and 5 energy units on the devices, respectively. Further, a time constraint is specified which requires that execution should finish within 10 time units. The following mappings of work packages onto devices are feasible and fulfill the time constraint:

- If we assign 4 work packages to the first device and 1 work package to the second device, the overall execution time equals to $max\{8, 5\} = 8$ units of time, while the overall energy consumption equals to $10 * 4 + 5 = 45$ units of energy consumption.
- If we assign 3 work packages to the first device and 2 work packages to the second device, the overall execution time equals to $max\{6, 10\} = 10$ units of time, while the energy consumption equals $10 * 3 + 5 * 2 = 40$ units.

Both work distributions satisfy the time constraint. However, the second work distribution is more energy-efficient than the first one.

6.2. Different Solutions Satisfying Optimality Criteria

6.2.1. Best performance

We have dealt outwardly in the previous chapters with distribution strategies that are directed towards best performance without taking into account the energy consumption. The basic idea, in this case, is to use all resources available in the cluster to achieve best performance. An exception, as discussed in Section 3.2, are some situations stemming from the dynamic on-demand work distribution, which might lead to suboptimal execution. In these cases a slower device might request one of the last work packages, delaying the overall execution since all over devices have finished processing. However, as we will see in the next chapter there exists methods to prevent such a scenario, especially using a static scheduling algorithm which would be fed with profiling data from the work package probe testing and avoid assigning work packages to the slowest devices at the end of execution.

6.2.2. Minimal energy

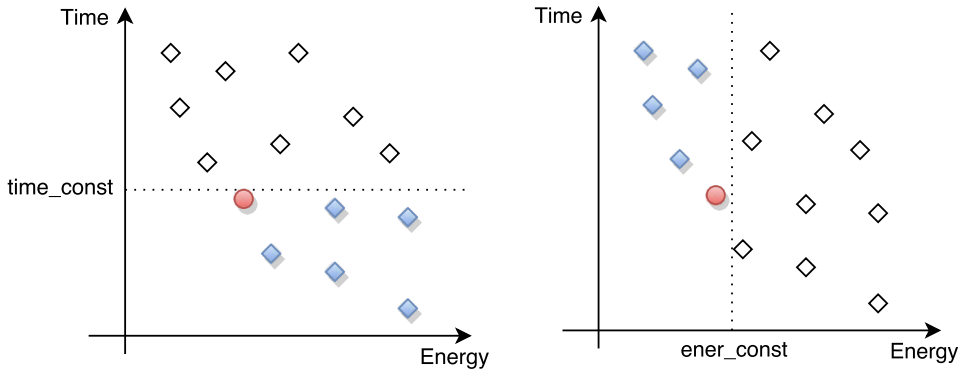
An optimization directed at achieving minimal energy consumption tends to ”serialize” the execution of an application. Minimal energy consumption means that only the most energy-efficient devices are used. As a consequence, in a heterogeneous environment only one type of devices with best energy-efficiency will be used. If only one instance of that device type exists in a cluster, the application will be executed on one device only.

6.2.3. Minimal energy with restrictions to performance

Since the minimal energy approach yields an extreme solution, there is a need for a relaxed approach. One possibility is to define a maximal execution time which shall be met with minimal energy preventing in this way the ”serializing” behavior. As shown in Figure 6.3a, the programmer sets a time constraint for which the most energy-efficient distribution shall be found. All distributions below the dotted line meet the time constraint with the circular point being the most energy-efficient one.

6.2.4. Best performance with restrictions to energy

Similar considerations as above lead to the definition of a maximal amount of energy which shall be met with best execution time possible. As shown in Figure 6.3b, the programmer sets an energy constraint for which the most time-efficient distribution shall be found. All distributions left to the dotted line meet the energy constraint with the circular point being the most time-efficient one.

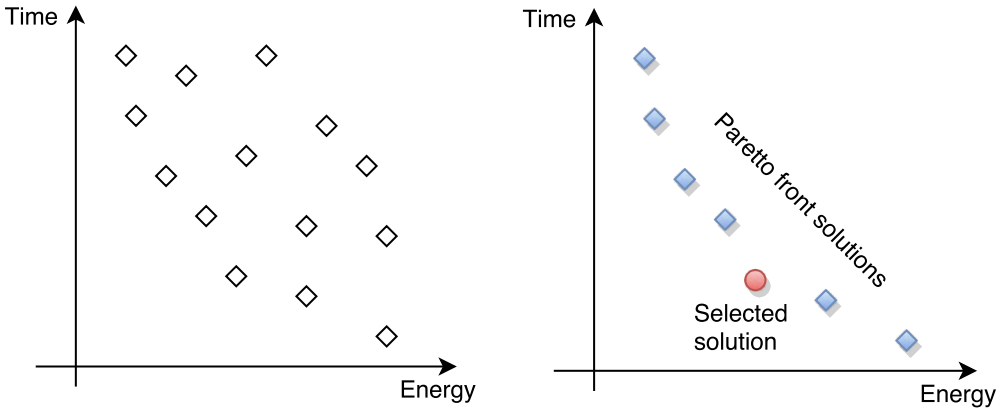


(a) Minimal energy with time constraint (b) Best performance with energy constraint

Figure 6.3.: Optimization problems with constraints

6.2.5. Time-energy trade-off.

Next, we address the problem that no constraints have been specified by the programmer. Our goal is to propose a ”good” solution by reasoning about time-energy trade-offs.



(a) All possible distributions (b) Pareto optimal distributions

Figure 6.4.: Set of possible solutions and Pareto front optimal solutions

Figure 6.4a shows the set of all possible distributions and Figure 6.4b the corresponding Pareto front. It is reasonable to assume that a distribution of the Pareto front with minimal distance from the origin is a good compromise between execution time and energy consumption–the selected distribution is highlighted with a circular point.

6.3. Summary

In this chapter, we discussed the problem of distributing work onto devices of a heterogeneous, asymmetric cluster when both execution time and energy consumption are taken into account. We presented different distribution strategies and discussed their implications in practice.

CHAPTER 7

Optimal Time and Energy Work Distributions in Heterogeneous Systems

Different strategies have been proposed for data-parallel HPC applications to tackle the problem of work distribution, from static hand-tuning to dynamic runtime techniques. HPC applications have the property that the data domain can be partitioned into equal-sized work packages and can be distributed to the parallel computational devices as outlined in Chapter 3 and 4. In this way, faster computational devices complete jobs faster than slower devices assuring a balanced work distribution.

The scheduling strategies for assigning work packages to computational devices can be guided by different optimization goals. Usually, the scheduling goal is to reduce execution time. However, for heterogeneous computational devices, it is not adequate to focus on execution time only. Another scheduling goal for assigning work packages is to reduce energy consumption. Unfortunately, runtime efficiency and energy efficiency are often conflicting objectives pushing in different directions, i.e. shorter execution time results in more energy consumption and a reduction in energy increases execution time.

The work packages are assigned to the devices by the dispatcher module. Different scheduling strategies can be realized by the dispatcher module. There are four scheduling strategies of special interest:

A The *most time-efficient solution* stressing runtime-efficiency only

B The *most energy-efficient solution* stressing energy-efficiency only

C The *most energy-efficient solution under a time constraint* enforcing a maximum makespan not to be exceeded

D The *most time-efficient solution under an energy constraint* enforcing a maximum energy limit not to be exceeded

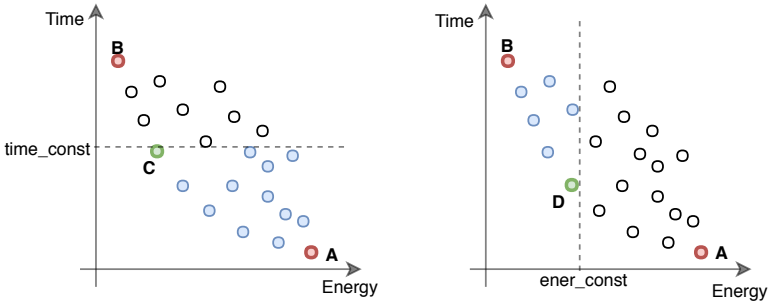


Figure 7.1.: Optimal solutions A, B, C, and D.

The four solutions A-D are illustrated in Figure 7.1. Each point in the diagram represents one mapping instance of work packages to compute devices with associated makespan and energy consumption. Solutions A and B are shown in both diagrams as lower and upper points, respectively. The left diagram illustrates solution C with time constraint *time_const*, while the right diagram solution D with energy constraint *ener_const*. The blue points represent in both diagrams valid solutions.

In this chapter, we discuss these four optimization problems in detail and present efficient algorithms which yield corresponding solutions for each of them. Whereas the two extrema A and B can be determined without any user input, C and D are based on the specification of constraints. For all optimization problems, we succeeded to compute optimal solutions.

Contributions.

We present four optimal algorithms that solve bi-criteria optimization problems in a Time-Energy solution space for work distribution based on our workload partitioning approach. More precisely, the contributions of this work are:

(a) An algorithm that finds the optimal distribution of work packages for best performance in a cluster setting, eliminating the work package assignment problems which appear with simple on-demand work distribution that delay the completion of the execution process,

(b) An algorithm which finds an optimal distribution of work to devices while guaranteeing minimal energy consumption,

(c) An algorithm which within a time constraint (runtime deadline) finds the best solution for work distribution which ensures minimal energy consumption,

(d) An algorithm which within an energy constrain (energy consumption budget) finds the best solution for work distribution which ensures best performance,

(e) Evaluation of algorithms using real-world HPC applications.

Related Work.

Computational clusters with heterogeneous devices introduce new challenges with respect to execution time and energy consumption: Not only one-dimensional optimizations in the direction of execution time or energy consumption are more intricate, in addition both dimensions have to be taken into account. We group the related work in three classes with the last one being closest to our work.

System level power management. Etinski et al. [ECLV12] use DVFS to affect CPU power consumption by running jobs with different CPU frequencies. Offline scheduling algorithms involving bi-objective optimization for uni-processor and multiprocessor systems incorporating speed scaling are considered in Bunde [Bun06]. Rozycki et al. [RW15] deal with power-aware scheduling on identical parallel machines, while Li [Li12] and Asghari et al. [AMW13] propose different strategies for balancing between energy consumption and performance on multi-core servers. Another approach by Lama et al. [LLA+13] focuses on turning on and off different nodes and migrating GPU workloads to achieve better power management utilizing the VOCL framework. However, this model is restricted to GPU clusters and it requires providing power usage data for different system configurations by the datacenter administrator. These research efforts differ considerably from ours since we are targeting applications.

Application level power management on homogeneous systems Freeh et al. [FLP+07] propose an approach in lowering energy consumption by dynamically adjusting power-performance states in high-performance cluster nodes. Gschwandtner et al. [GDF14] propose a method RS-GDE3 based on an evolutionary algorithm to tune HPC codes for multi-processor systems. The main difference to our work is that we address heterogeneous systems instead of homogeneous ones.

Application level power management on heterogeneous systems Wang et al. [WR10] propose an approach of inter-processor work distribution using processor frequency scaling to minimize energy consumption in a CPU-GPU heterogeneous system. Balaprakash et al. [BTW14] focus on bi-criteria optimization between time and energy or time and power for HPC codes in multi-processor systems and Intel Xeon Phi. Work of Tarplee et al. [TFMS15] deals with generating Pareto front solutions for bi-criteria optimization similar to our trade-off solutions but does not address finding solutions under constraints efficiently.

The main difference between our approach and the research efforts cited above is that we introduce time constraints and energy constraints, discuss op-

timality for executing applications and try to find optimal solutions efficiently.

7.1. Optimization Model

In this chapter, we present our optimization model and give a formal definition of our optimization problem:

1. The input data space is split into a set of equal-sized work packages **W** with cardinality n.
2. The hardware system consists of a set of heterogeneous devices **D** with cardinality m.
3. Each work package $w \in \mathbf{W}$ is assigned to a device $d \in \mathbf{D}$.
4. Each device $d \in \mathbf{D}$ has associated a pair (t_d, e_d) with
   t_d ... specifying units of time required to process one work package on device d, and
   e_d ... specifying units of energy required to process one work package on device d
5. The number of work packages assigned to device d is denoted by x_d and is the solution of our optimization problems.

In our optimization model, we only count devices which are used during the execution of the application under consideration. Compute devices which are not used can be switched off or put in an idle state or used by another application depending on the support of the underlying hardware and operating system.

7.1.1. Energy measurement functions

Whereas measuring execution time is well supported, the same does not apply for determining energy consumption. The most accurate way to do such measurements is by installing energy meters adjacent and connected to each of the devices of the cluster. Since this is not practicable in many cases, we calculate energy consumption by an aggregate function which sums up power readings at several steps during the execution time of a work-package. These power readings are done using functions provided in manufacturer APIs which support querying for different device states and parameters. The energy consumption for a device is calculated by

$$e_d = \sum_i \frac{P_i + P_{i+1}}{2} * \Delta t, \quad \forall d \in D \tag{7.1}$$

In formula (7.1), P_i stands for the power draw from the device (measured in Watt) in the moment of the measurement, while Δt is the time between two measurements. Considering that the power draw fluctuates during the computation process reflecting the utilization of the device by the kernel code and data, more frequent power draw measurements will result in more accurate data, i.e. the accuracy can be improved by setting Δt to a smaller value.

Alternatively, values for t_d and e_d can be predicted as well.

7.2. Optimal Solutions

7.2.1. Most Time-Efficient Solution Without Constraint

Let $k_{wd} \in \{0, 1\}$ be a binary variable which is set to 1 if package w is assigned to device d, 0 otherwise. Thus the optimization problem for the most time-efficient *solution A* without energy constraint can be formulated as follows:

$$minimize \quad \max_{d \in D}(t_d * \sum_{w \in W} k_{wd}) \tag{7.2}$$

with the following condition

$$x_d = \sum_{w=1}^{n} k_{wd} \tag{7.3}$$

$$\sum_{w=1}^{n} \sum_{d=1}^{m} k_{wd} = n \tag{7.4}$$

ensuring that each work package is assigned to exactly one compute device.

This optimization problem can be solved efficiently by an incremental strategy: Look for the most time-efficient device and assign work packages to it until the accumulated execution times allow assignment to the second device or third device, and so on. Algorithm (1) computes the *optimal* solution for that optimization problem.

The algorithm takes as input work package processing times for each of the m devices $T = \langle t_1, t_2, \ldots, t_m \rangle$. The entries of array $X = \langle x_1, x_2, \ldots, x_m \rangle$ denote the same devices as T and are initialized to 0. A mapping of a work package to a device is carried out if the accumulated execution time plus the processing time for the work package is less than all other devices guaranteeing in this way optimality. The optimality can be proved by showing that the loop invariant "minimal execution time for assigned work packages" is maintained over all iterations. The invariant is trivially true for assigning the first work package to the fastest device. The invariant is also true for subsequent iteration as well, since the algorithm takes all possibilities into account for assigning the next work package. The algorithm calculates sequence X with the number of work packages assigned to the corresponding devices.

Algorithm 1: Most time-efficient solution A without constraint

input : time values $T = \langle t_1, t_2, \ldots, t_m \rangle$; number of work packages n; number of devices m
output: result mapping $X = \langle x_1, x_2, \ldots, x_m \rangle$

```
1  function A(T, n, m)
       // create array x with m elements initialized to 0
2      min ← 1 while n > 0 do
3          for i ← 2, m do
4              if (x_min + 1) * t_min > (x_i + 1) * t_i then
5                  min ← i
6              end
7          end
8          n ← n − 1 x_min ← x_min + 1
9      end
10     return X ;
```

7.2.2. Most Energy-Efficient Solution Without Constraint

The most energy-efficient *solution B* without time constraint can be found quite easily. This optimization problem reduces basically to the demand that work packages must be mapped to the most energy-efficient devices only. The optimality can be proved quite easily by showing that the sum of the consumed energy is minimal, if the individual terms are minimal. Let $D^E \subseteq D$ denote the subset of most energy-efficient devices, i.e. for all d, d'

$$(e_d = e_{d'} | d, d' \in D^E) \wedge (e_d < e_{d'} | d \in D^E, d' \in D \setminus D^E) \tag{7.5}$$

Instead of a single optimal solution, we have a set of optimal solutions: Each element of the power set of D^E can be used as a possible mapping target for work packages. One extreme case is to use only one device from the set D^E while the other extreme case results from splitting the work packages equally between all the devices in set D^E decreasing the overall execution time proportional to the cardinality of set D^E, which consists of devices with same characteristics, i.e.

$$x_d = n / \left| D^E \right| \tag{7.6}$$

where x_d is the number of work packages mapped to device d and for the ease of presentation ignoring a possible remainder which is straight-forward to be handled.

7.2.3. Most Energy-Efficient Solution Under a Time Constraint

Let T_c denote a time constraint—an upper time limit which shall not be exceeded. With x_d denoting the number of work-packages assigned to device d, the optimization problem of the most energy-efficient *solution* C under a time constraint can be formulated as follows:

$$x_d * t_d \leq T_c, \quad \forall d \in D \tag{7.7}$$

whereby the energy consumption shall be minimal

$$minimize \quad \sum_{d=1}^{m} (x_d * e_d) \tag{7.8}$$

with the following condition

$$\sum_{d=1}^{m} x_d = n \tag{7.9}$$

This optimization problem can be solved efficiently by a greedy approach. The devices are sorted with the most energy-efficient devices first. Subsequently, work packages are mapped to those devices in ascending order obeying formula (7.7) until all work packages have been assigned or the time constraint cannot be fulfilled. Algorithm 2 computes the *optimal* solution if solvable, and error code -1 otherwise. The input of the function is a sequence of time and energy values for each of the devices of the targeted cluster.

The algorithm takes as input time values $T = \langle t_1, t_2, \ldots, t_m \rangle$ denoting the processing times for work packages for the devices, and energy values $E = \langle e_1, e_2, \ldots, e_m \rangle$ denoting the energy consumption. E is sorted with the most energy-efficient device first denoted by $\langle e'_1, e'_2, \ldots, e'_m \rangle$ with $e'_1 \leq e'_2 \leq \ldots \leq e'_m$. T is reordered accordingly such that at the same position in E and T the very same device is referred to. The same applies to sequence $X = \langle x'_1, x'_2, \ldots, x'_m \rangle$ where each of the entries correspond to the same devices as in E and T.

Algorithm (2) starts from the most energy-efficient devices and assigns as many work packages as possible that can be processed under the given time constraint achieving in this way, an optimal solution. Since the algorithm obviously obeys the time constraint, a proof only has to show that the solution is the most energy-efficient one. As the devices with minimal energy consumption are taken first, it can be shown that the sum of the consumed energy is minimal as well. However, usually the number of work packages created by the partitioning process will not amount exactly to the full processing capacity of the available devices. Thus, two cases must be handled:

Algorithm 2: Most energy-efficient solution C under a time constraint

```
   input  : time values T = ⟨t_1, t_2, ..., t_m⟩, energy values
            E = ⟨e_1, e_2, ..., e_m⟩; number of work packages n; time
            constraint T_c
   output: result mapping X = ⟨x'_1, x'_2, ..., x'_m⟩
 1 function C(T, E, n, T_c)
       // sort E resulting in ⟨e'_1, e'_2, ..., e'_m⟩
       // reorder T resulting in ⟨t'_1, t'_2, ..., t'_m⟩
       // create array X with m elements initialized to 0
 2     d ← 0
 3     while n > 0 do
 4         if d > m then
 5             return −1 // run out of devices
 6         end
 7     end
 8     if ⌊T_c/t_d⌋ < n then
 9         x_d ← ⌊T_c/t_d⌋
10     end
11     else
12         x_d ← n
13     end
14     n ← n − x_d;
15     d ← d + 1
16     return X ;
17 end
```

1. *Number of work packages exceeds the processing capacity of the available devices.*

   In this case, no solution exists which fulfills the constraint. For a solution the time constraint must be relaxed by increasing the execution time.

2. *Not all devices get work packages assigned.*

   Since it is a greedy algorithm, it may happen that all work packages have been assigned and there are devices left over for which no work is available anymore. This does not harm the optimality criteria and it is an optimal solution. However, there exist further optimal solutions. If the number of work packages assigned to devices of the same type differs, reassignment of work packages within that device type class enforces an equal distribution and represents an optimal solution as well. If the time constraint can be fulfilled by using devices of only one device type, reassignment will result in faster execution times. In that case, it can be chosen between two different solutions, one solution which uses fewer cluster resources but

takes longer to finish, and the other solution which uses more cluster resources but executes faster. Both solutions fulfill the optimality criteria and it depends on usage policy of the cluster which one is preferable.

7.2.4. Most Time-Efficient Solution Under an Energy Constraint

In order to generate a mapping plan which delivers the most time-efficient *solution* D under an energy constraint the following must hold:

$$\sum_{d=1}^{m} x_d * e_d \leq E_c \tag{7.10}$$

whereby the execution time shall be minimal

$$minimize \quad \max_{d \in D}(x_d * t_d) \tag{7.11}$$

with the following condition

$$\sum_{d=1}^{m} x_d = n \tag{7.12}$$

with x_d denoting the number of work packages assigned to device d.

Unlike the problems above, a solution to this problem can not be found by step-wise assigning work packages to devices, since the final mapping must satisfy the energy constraint as indicated by formula (7.10).

A simple algorithm for finding an optimal solution for this problem is to evaluate all possible mappings of n work packages to m devices. Given that the effort required to process work packages does not differ on identical devices, the number of possible mappings which have to be tested equals to

$$\binom{n+m-1}{m-1}$$

resulting in an intractable approach.

We can express our problem as the following mathematical program:

$$\begin{array}{lll} \text{minimize} & \max_{d \in D} t_d \cdot x_d & \\ \text{subject to} & \sum_{d \in D} e_d \cdot x_d \leq E_c & \\ & \sum_{d \in D} x_d = n & \\ & x_d \in \mathbb{N}_0 & \forall d \in D \end{array} \tag{7.13}$$

where x_d are integer variables denoting the number of work packages performed by each device. The first constraint ensures that the energy consumption is limited by total energy capacity E_c. The second constraint ensures that all work packages n are alloted to the devices. The last constraint is the integrality

constraint of variable x_d and ensures that work packages are allocated as a whole to devices. A solution x to the mathematical program is called a *schedule*.

We can rewrite the above as an integer linear program as follows,

$$\begin{array}{lll} \text{minimize} & \lambda & \\ \text{subject to} & \sum_{d \in D} e_d \cdot x_d \leq E_c & \\ & \sum_{d \in D} x_d = n & \\ & t_d \cdot x_d \leq \lambda & \forall d \in D \\ & x_d \in \mathbb{N}_0 & \forall d \in D \\ & \lambda \in \mathbb{R}_0^+ & \end{array} \tag{7.14}$$

where λ is the makespan of the schedule.

Optimal Bisection Algorithm

In this section, we develop an algorithm which solves Equation 7.14 optimally. Instead of tackling the problem of minimizing makespan subject to a bound on energy consumption directly, we apply a simple variant of a framework of Hochbaum and Shmoys [HS87] which lets us reduce our original problem to the "dual" problem of minimizing energy consumption subject to a bound on makespan. As we will see later, the dual problem is a simple packing problem that is amenable to a greedy algorithm.

In general, given an optimization problem with two parameters—in our case, energy consumption and makespan—there are two optimization problems that can be obtained by optimizing one of the parameters subject to a constraint on the other. One of the problems is called the *primal* problem and the other is called the *dual* problem. The framework of Hochbaum and Shmoys [HS87] allows one to use an exact algorithm for the dual problem to obtain an exact algorithm for the primal (and vice versa) using bisection[a]. Let us now explain how to use this framework for our purposes.

For our setting, the primal problem is minimizing makespan subject to a bound on energy consumption while the dual problem is minimizing energy consumption subject to a bound on makespan λ. Let $M(\lambda)$ be the dual problem, i.e. minimum energy consumption for the makespan parameter λ. If there does not exist a schedule which satisfies makespan λ, for example, when $\lambda < \min_{d \in D} t_d$, then $M(\lambda)$ is set to ∞.

[a]The bisection (also known as interval halving or dichotomy) method has been used in mathematics to find roots of a function. The method repeatedly bisects an interval and chooses a sub-interval in which the root must lie. The intervals from which the methods start from, and the chosen sub-intervals must have the property that one of the endpoints is positive and the other one negative, i.e., a sufficient condition for the existence of a root in the interval [BF11].

Our original problem can be rewritten as:

$$\begin{array}{ll} \text{minimize} & \lambda \\ \text{subject to} & M(\lambda) \leq E_c \\ & \lambda \in \mathbb{R}_0^+ \end{array} \tag{7.15}$$

Thus, given an efficient procedure for computing $M(\lambda)$, we can determine the optimal value of λ for our original problem using bi-section on the search space of λ.

Computing $M(\lambda)$

Now we show how to compute $M(\lambda)$. Observe that a feasible schedule can only assign at most $\lfloor \frac{\lambda}{t_d} \rfloor$ packages to device $d \in D$. Thus, we can express the problem of finding a feasible schedule that minimizes energy consumption as the following ILP:

$$\begin{array}{lll} \text{minimize} & \sum_{d \in D} e_d \cdot x_d & \\ \text{subject to} & \sum_{d \in D} x_d = n & \\ & x_d \leq \lfloor \frac{\lambda}{t_d} \rfloor & \forall d \in D \\ & x_d \in \mathbb{N}_0 & \forall d \in D \end{array} \tag{7.16}$$

With a simple greedy algorithm, an optimal solution for $M(\lambda)$ can be computed. Suppose that the devices are sorted in ascending order of energy costs and let t_i and e_i be the processing time and energy cost of the i-th device in this order. The greedy algorithm iterates through the devices in this order and assigns as many unassigned work packages to the current device d as possible (up to $\lfloor \frac{\lambda}{t_d} \rfloor$ work packages) until all work packages are assigned. If it manages to assign all work packages, it returns the energy consumption of this schedule; otherwise, it returns ∞. See Algorithm 3 for a detailed description of the algorithm.

Lemma 1. *Algorithm 3 returns $E_t = M(\lambda)$.*

Proof. As we observed earlier, a schedule with makespan at most λ schedules at most $\lfloor \frac{\lambda}{t_d} \rfloor$ work packages on device $d \in D$. Thus, there exists a feasible schedule with makespan at most λ if and only if $\sum_d \lfloor \frac{\lambda}{t_d} \rfloor \geq n$. Since Algorithm 3 packs as many unassigned work packages onto each device that it considers, it returns $E_t = \infty$ if and only if $\sum_d \lfloor \frac{\lambda}{t_d} \rfloor < n$. Therefore, it returns $E_t = M(\lambda)$ in the case that there is no feasible schedule with makespan at most λ. In the remainder of the proof, we consider the case that there is a feasible schedule with makespan at most λ.

Let x_i be the number of work packages assigned to device i by Algorithm 3. We have that x is a feasible solution to ILP (7.16) by construction. Since

Algorithm 3: Packing algorithm for makespan λ

```
1  function M(λ)
       // Number of packages left to execute on remaining
          devices
2      w ← n;
       // Set i to first device
3      i ← 1;
       // Set total energy consumption E_t to zero
4      E_t ← 0;
5      while i ≤ m ∧ w > 0 do
           // compute number of packages on device i
6          x_i ← min{⌊λ / t_i⌋, w} ;
           // update energy consumption
7          E_t ← E_t + x_i · e_i ;
           // update remaining work
8          w ← w − b ;
           // switch to next device
9          i ← i + 1;
10     end
11     if w > 0 then
           // Makespan insufficient for packing
12         E_t ← ∞ ;
13     end
14     return E_t ;
```

$E_t = \sum_{i=1}^{m} x_i \cdot e_i$, it remains to prove that any other feasible solution $y \neq x$ to ILP (7.16) has energy consumption $\sum_{i=1}^{m} y_i \cdot e_i \geq \sum_{i=1}^{m} x_i \cdot e_i$. To do this, we use a greedy exchange argument (cf. [KT06]), that is, we show that there is a transformation ϕ that transforms y to another feasible solution $z = \phi(y)$ such that z is "closer" to x and $\sum_{i=1}^{m} y_i \cdot e_i \geq \sum_{i=1}^{m} z_i \cdot e_i$, where the distance between any two solutions a and b is defined to be

$$\text{dist}(a, b) = \sum_{i=1}^{m} |a_i - b_i|.$$

Let us first see how to finish the proof given such a transformation ϕ. Define $z(0) = y$ and $z(j) = \phi(z(j-1))$. We have that $\text{dist}(x, y) \leq 2n$ because each feasible solution schedules exactly n work packages in total. Since $\text{dist}(x, z(j)) < \text{dist}(x, z(j-1))$, for some $k \leq n$, we have $\text{dist}(x, z(k)) = 0$, and thus $z(k) = x$. Moreover, $z(j)$'s energy consumption is at most that of $z(j-1)$ for each $0 < j \leq k$. Thus, we get that the energy consumption of $x = z(k)$ is

at most that of $y = z(0)$, as desired.

We now turn to proving the existence of $z = \phi(y)$ that satisfies the above properties. Define $i_{\min} = \min\{i : x_i \neq y_i\}$ and $i_{\max} = \max\{i : x_i \neq y_i\}$, i.e. device $i_{\min}$ ($i_{\max}$, resp.) is the one with the least (most, resp.) energy cost among the devices to which x and y assigns different numbers of work packages. Since $x \neq y$ and both assign exactly n work packages to devices, we have that $i_{\min} < i_{\max}$. Moreover, Algorithm 3 assigns as many work packages as it can to the devices with lower energy costs, so we have

$$x_{i_{\min}} \geq y_{i_{\min}} + 1 \tag{7.17}$$

and

$$x_{i_{\max}} \leq y_{i_{\max}} - 1. \tag{7.18}$$

Now, consider the solution z defined as follows:

$$z_i = \begin{cases} y_i & \text{for } i \notin \{i_{\min}, i_{\max}\} \\ y_i + 1 & \text{for } i = i_{\min} \\ y_i - 1 & \text{for } i = i_{\max} \end{cases}$$

Essentially, z is obtained from y by taking a work package that is assigned to device $i_{\max}$ and assigning it to device $i_{\min}$. First, we show that z is a feasible solution to ILP (7.16). By Inequality (7.18) and the fact that x is a feasible solution, we have that $z_{i_{\min}} = y_{i_{\min}} + 1 \leq x_{i_{\min}} \leq \lfloor \frac{\lambda}{t_{i_{\min}}} \rfloor$. For $i \neq i_{\min}$, we have that $z_i \leq y_i \leq \lfloor \frac{\lambda}{t_i} \rfloor$ since y is a feasible solution. Clearly, z also assigns a total of n work packages. Thus, z is indeed a feasible solution. Second, we have that $\sum_{i=1}^{m} y_i \cdot e_i \geq \sum_{i=1}^{m} z_i \cdot e_i$ because z is obtained from y by taking a work package that is assigned to a device with energy cost $e_{i_{\max}}$ and assigning it to a device with energy cost $e_{i_{\min}} \leq e_{i_{\max}}$ (because $i_{\min} < i_{\max}$). Finally, we show that $\text{dist}(x, z) < \text{dist}(x, y)$. By construction of z, we have

$$|x_i - z_i| = \begin{cases} |x_i - y_i| & \text{for } i \notin \{i_{\min}, i_{\max}\} \\ |x_i - y_i| - 1 & \text{for } i = i_{\min} \\ |x_i - y_i| - 1 & \text{for } i = i_{\max} \end{cases}$$

where the last two equalities are due to Inequalities (7.17) and (7.18), respectively. Therefore, we have $\text{dist}(x, z) < \text{dist}(x, y)$. □

Lemma 2. *Function $M(\lambda)$ is a monotonic decreasing function for a given problem instance, i.e.,*

$$\forall \lambda_1, \lambda_2 \in \mathbb{R}_0^+ : \lambda_1 \leq \lambda_2 \Rightarrow M(\lambda_1) \geq M(\lambda_2). \tag{7.19}$$

Proof. The lemma statement trivially holds if $M(\lambda_1) = \infty$. Suppose $M(\lambda_1) <$

∞. Let x_1^* and x_2^* be optimal solutions to ILP (7.16) with makespan parameter λ_1 and λ_2, respectively. We have that $M(\lambda_1)$ and $M(\lambda_2)$ are the energy consumptions of x_1^* and x_2^*, respectively. Since $\lambda_1 \leq \lambda_2$, we have that x_1^* is also a feasible solution to ILP (7.16) with makespan parameter λ_2. Therefore, by optimality of x_2^* for ILP (7.16) with makespan parameter λ_2, we have that $M(\lambda_1) \geq M(\lambda_2)$. □

Bisection Approach and Makespan Bounds

After solving the dual problem, which is a prerequisite for applying the bisection approach, we proceed with the bisection technique. In each bisection step, we assume that the lower-bound of the makespan parameter constitutes an infeasible solution, i.e. $M(\lambda) > E_c$, and the upper-bound constitutes a feasible solution, i.e. $M(\lambda) \leq E_c$. Thus, we can find the boundary between an infeasible and feasible solution that represents an optimal solution for λ. Since processing times t_d are assumed to be integers, the optimal λ is also an integer, and so the total number of bisection steps is at most logarithmic in the size of the search space of λ. The boundary can be perceived as the root which is found by our adapted bisection method.

We now derive lower and upper bounds on the parameter λ to be used by the bisection algorithm. A lower bound on λ can be deduced by an averaging argument as follows. Suppose x is a feasible schedule with makespan at most λ. On the one hand, we have $x_d \leq \left\lfloor \frac{\lambda}{t_d} \right\rfloor$ for all $d \in D$. Applying the triangularity inequality, we obtain: $\sum_{d \in D} x_d \leq \sum_{d \in D} \left\lfloor \frac{\lambda}{t_d} \right\rfloor$. On the other hand, since the packing equality $\sum_{d \in D} x_d = n$ must hold, we obtain $n \leq \sum_{d \in D} \left\lfloor \frac{\lambda}{t_d} \right\rfloor \leq \sum_{d \in D} \frac{\lambda}{t_d}$. Hence, we obtain the following lower bound for the parametric makespan λ,

$$\frac{n}{\sum_{d \in D} \frac{1}{t_d}} \leq \lambda.$$

The largest makespan that the bisection algorithm needs to consider is that of the schedule that minimizes energy consumption. Such a schedule assigns all n work packages to the device with the cheapest energy cost. Recall that we sorted the devices in non-decreasing order of energy cost. Thus, t_1 is the processing time of the device with the cheapest energy cost, and the makespan of the schedule that minimizes energy consumption is $n \cdot t_1$. Hence, we conclude with the following upper and lower bounds on λ:

$$\frac{n}{\sum_{d \in D} \frac{1}{t_d}} \leq \lambda \leq n \cdot t_1. \tag{7.20}$$

Consider Figure 7.2 for illustration. The initial lower and upper bounds are given by $\langle \lambda_1, \lambda_2 \rangle$ with λ_1 being an infeasible solution and λ_2 a feasible one. Since mid-point λ_3 is a feasible solution, the new lower and upper bounds are

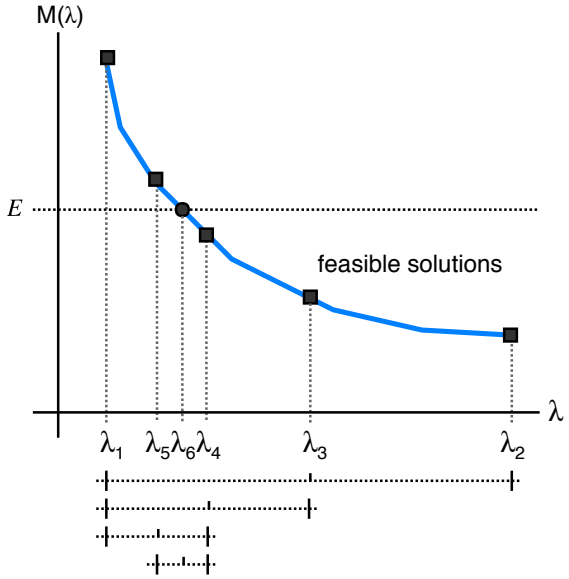


Figure 7.2.: Bisection Approach

$\langle \lambda_1, \lambda_3 \rangle$. The next mid-point λ_4 is feasible as well yielding $\langle \lambda_1, \lambda_4 \rangle$. Mid-point λ_5, however, is infeasible resulting in $\langle \lambda_5, \lambda_4 \rangle$, narrowing in this way the given global energy budget of E.

Bisection Algorithm

The optimization problem of Equation (7.15) can be solved in log number of invocations of function $M(\lambda)$ using a bi-section algorithm as outlined in Algorithm 4. If the lower bound is a feasible solution, the algorithm does not perform bi-section and terminates with the lower bound as an optimal solution for λ. If it is an infeasible solution, a feasible upper-bound is assumed and the bi-section is performed. In each iteration the invariant holds that the lower-bound is an infeasible solution and the upper bound is a feasible solution.

The running time of our algorithm consists of sorting the devices according to their energy costs and the bisection algorithm. Sorting takes $O(m \log m)$ time. Let us now analyze the running time of the bisection algorithm. First, we deduce an upper bound on the number of iterations k performed by the bi-section algorithm using the lower and upper bound of Inequality (7.20). Since we assumed integer values for processing times and the search range halves in each iteration, we have that

$$k \leq \left\lceil \log_2 n + \log_2 \left(t_1 - \frac{1}{\sum_{d \in D} \frac{1}{t_d}} \right) \right\rceil \leq \lceil \log_2 n \rceil + \lceil \log_2 t_1 \rceil \,.$$

Each iteration of the bisection algorithm consists of a single invocation of Algorithm 3 which takes $O(m)$ time since it involves packing at most m devices. Therefore, the total time taken is

$$O(m(\log m + \log n + \log t_1)).$$

Algorithm 4: Most time-efficient solution D under an energy constraint

```
1  function D(E_c)
2      λ_l ← ⌊ n / (Σ_{i=1}^{m} 1/t_i) ⌋;
3      if M(λ_l) > E_c then
           // Lower-bound is infeasible; upper-bound is feasible
           // This invariant stays intact in each iteration
4          λ_u ← n · t_1;
5          while λ_u − λ_l > 1 do
               // Compute mid-point
6              λ_m ← ⌊ (λ_u − λ_l)/2 ⌋;
7              if M(λ_m) ≤ E_c then
                   // Mid-point is feasible
8                  λ_u ← λ_m ;
9              else
                   // Mid-point is infeasible
10                 λ_l ← λ_m ;
11             end
12         end
           // Return upper-bound as optimal solution
13         return ⟨λ_u, M(λ_u)⟩
14     else
           // Return lower-bound as optimal solution
15         return ⟨λ_l, M(λ_l)⟩
16     end
```

7.3. Experimental Evaluation

In this section, we present the results from our experiments regarding the feasibility of the proposed work distribution solutions in balancing between execution time and energy consumption in a heterogeneous asymmetric cluster.

We have tested our optimization approaches in our distributed-memory clusters PHIA and EXA. Hardware characteristics of those clusters have been sum-

marized in Table 5.2. Experiments are run using PSC6, PAC1 and EAC1 cluster configurations which are detailed in Table 5.3. PSC6 consists of six PHIA nodes with 12 nVidia Tesla K20m and 12 Xeon Phi devices distributed evenly in a symmetrical arrangement. PAC1 consists of three PHIA nodes with 6 nVidia Tesla K20m and 6 Xeon Phi distributed unevenly in an asymmetrical arrangement. EAC1 consists of two EXA nodes each with one device, a nVidia Tesla V100 and an AMD Radeon Instict MI25 respectively.

Our experiments are based on our benchmark suite, which is detailed in Table 5.1.

7.3.1. Optimal solutions A, B, C and D

We first investigate the optimal solutions for a synthetic data-parallel program. It is a single kernel application consisting of a stencil code. This experiment is done using all 8 PHIA nodes in an asymmetric arrangement. In Figure 7.3, we have shown the overall execution time and energy consumption for each of the optimization problems A, B, C, and D.

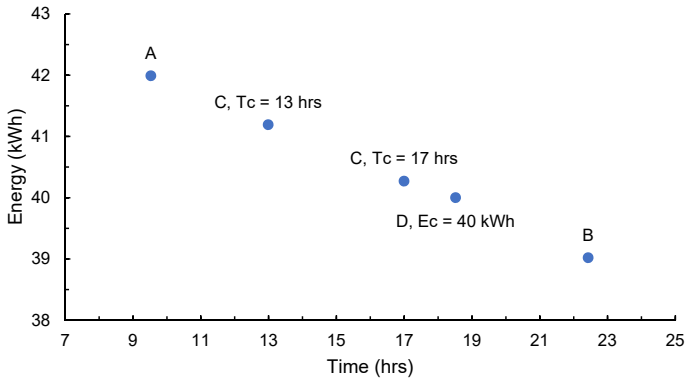
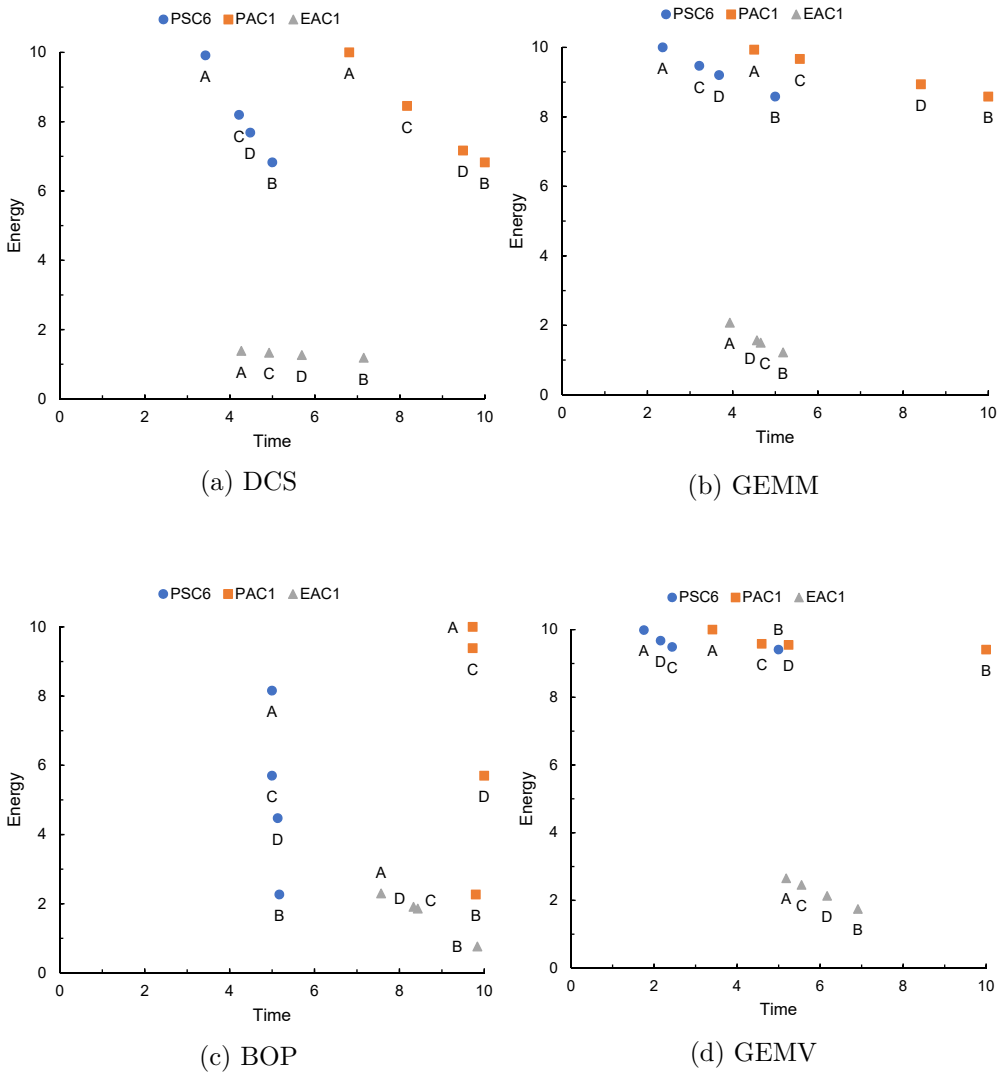


Figure 7.3.: Optimal Work Distributions for a Synthetic Parallel Program

Fastest execution using algorithm A takes 9.53hrs, and the required energy consumption is 42kWh. On the other hand, aiming for minimal energy consumption using algorithm B reduces energy consumption to 39kWh but extends execution to 22.5hrs. In this particular case, distribution strategies are more sensitive towards runtime where for small reductions in energy consumption, the execution is delayed significantly. As we will see with other experiments, sensitivity towards runtime or energy consumption depends on the application and hardware arrangements. Using algorithms with constraints, we can mitigate this problem by setting deadlines (runtime constraints) or energy budgets which must not be exceeded. For instance, for this application a time constraint can be set at 13hrs, which using algorithm C can ensure that the execution will be finished within that deadline with minimal energy consumption for such settings - 41.19kWh. Alternatively, a user preference can be to complete execution

as soon as possible but without exceeding an energy budget of 40kWh. Minimal execution time obeying the energy constraint, in this case, is 18.52hrs.

In the following experiment, we calculate optimal solutions A, B, C and D for selected HPC applications DCS, GEMM, BOP, and GEMV.



(a) DCS

(b) GEMM

(c) BOP

(d) GEMV

Figure 7.4.: Optimal Work Distribution with Algorithms A, B, C and D

In Figure 7.4, we show the results of this experiment in normalized time and energy units. Optimal solutions C and D are calculated using arbitrary runtime and energy constraints within the bounds of solutions A and B. In Tables 7.1 - 7.4, we present detailed experimental data comparing the share of processed work packages between devices with the share of energy consumption by these devices. We have omitted the experimental data for solution B since it only uses

the most energy-efficient devices, therefore work share and energy consumption share would be 100% in all cases. We also omitted PAC1 experimental data, since they are similar to PSC6 data, and both configurations use the same devices.

In Figure 7.4a, we present all optimal solutions for PSC6, PAC1, and EAC1 for DCS. We note that for a PSC6 cluster configuration, optimal solutions are more sensitive towards energy consumption. Solutions A and B are 3 energy units and 1.5 time units away from each other. Moving towards a faster execution costs 50% more in energy consumption per each time unit. In PAC1, we see a similar distribution of solutions. However, the most noticeable change is with EAC1 configuration, where the execution time can be halved without a remarkable change in energy consumption. In this configuration, only two devices are used and as shown in Table 7.1, V100 is almost twice as fast and slightly more energy-efficient than MI25. As we move from solution B which uses only V100 to solution A, work packages are shifted towards MI25 reducing significantly overall execution time, while overall energy consumption is increased only by the difference in energy consumption between the devices.

Table 7.1.: Work vs. Energy Consumption Share: DCS

		Work Packages (%) / Energy Consumption Share (%)		
		A	**C**	**D**
PSC6	K20m	67.86% / 46.70%	85.71% / 71.35%	91.07% / 80.90%
	Phi	32.14% / 53.30%	14.29% / 28.65%	8.93% / 19.10%
EAC1	V100	64.29% / 55.02%	74.11% / 66.04%	85.71% / 80.31%
	MI25	35.71% / 44.98%	25.89% / 33.96%	14.29% / 19.69%

In Figure 7.4b, we see that for GEMM solutions are more runtime sensitive. For PSC6 and PAC1 we observe the same pattern where the difference between solution A and B in terms of runtime are big. In both cases, the runtime can be halved without a noticeable change in energy consumption. We use 24 devices in PSC6 and 12 devices in PAC1, half of them are K20m and the other half are Xeon Phi. By observing experimental data in Table 7.2, we see that Xeon Phi is faster than K20m as reflected by the work share, however less energy-efficient than K20m. For solution A, Xeon Phi takes a bigger share of work. However, since K20m is more energy-efficient, algorithm B executes all work packages in K20m devices providing a solution with minimal energy consumption but with a significant slowdown of execution. EAC1 configuration has a balanced distribution of solutions.

In Figure 7.4c, we see the opposite case where for BOP, solutions for cluster configurations PSC6 and PAC1 are highly sensitive to energy consumption.

Table 7.2.: Work vs. Energy Consumption Share: GEMM

		Work Packages (%) / Energy Consumption Share (%)		
		A	**C**	**D**
PSC6	K20m	42.86% / 36.78%	64.29% / 58.27%	75.00% / 69.95%
	Phi	57.14% / 63.22%	35.71% / 41.73%	25.00% / 30.05%
EAC1	V100	78.13% / 45.87%	92.86% / 75.53%	91.07% / 70.77%
	MI25	21.88% / 54.13%	7.14% / 24.47%	8.93% / 29.23%

Table 7.3.: Work vs. Energy Consumption Share: BOP

		Work Packages (%) / Energy Consumption Share (%)		
		A	**C**	**D**
PSC6	K20m	97.32% / 54.03%	99.11% / 78.73%	100% / 100%
	Phi	2.68% / 45.97%	0.89% / 21.27%	0% / 0%
EAC1	V100	16.52% / 44.51%	14.29% / 47.54%	8.26% / 91.74%
	MI25	83.48% / 55.49%	85.71% / 52.46%	26.74% / 73.26%

As backed by experimental data in Table 7.3, from the work and energy consumption share between devices, we can extrapolate that Xeon Phi is highly inefficient both in terms of execution time and energy consumption. However, in case of algorithm A no constraints on energy consumption can be set, therefore the algorithm assigns few work packages to Xeon Phi which slightly reduces execution time while significantly increasing energy consumption compared to solution B. EAC1 configuration does not reflect this sensitivity since the discrepancies in execution time and energy consumption between the devices are minimal leading to a balanced distribution of solutions.

Table 7.4.: Work vs. Energy Consumption Share: GEMV

		Work Packages (%) / Energy Consumption Share (%)		
		A	**C**	**D**
PSC6	K20m	32.14% / 35.95%	3.57% / 4.20%	14.29% / 16.49%
	Phi	67.86% / 64.05%	96.43% / 95.80%	85.71% / 83.51%
EAC1	V100	75.00% / 49.38%	80.36% / 57.09%	89.29% / 73.05%
	MI25	25.00% / 50.62%	19.64% / 42.91%	10.71% / 26.95%

In Figure 7.4d for GEMV we see a similar situation as in the case of GEMM for all cluster configurations. The only exception as we can observe from experimental data in Table 7.4 is that Xeon Phi is much faster and more energy-efficient than K20m compared to the tests run for GEMM.

7.4. Summary

This chapter addressed the problem of distributing work onto heterogeneous devices, whereby both execution time and energy consumption are taken into account. We defined four different problems A–D which are of particular interest for users. We succeeded to provide optimal solutions for all of the optimizations problems. Moreover, as shown in experimental section for different cluster configurations, the combined approach of workload partitioning into same-size medium-grain work packages and use of algorithms A–D for mapping work packages to devices reduces the complexity of work distribution in asymmetric clusters requiring equivalent effort as for work distribution in symmetric clusters.

CHAPTER 8

Analysis of the Time/Energy Solution Space For Work Distribution

In the previous chapter, we presented various solutions for our two-dimensional optimization problem. We started with the two extrema A and B which span the two-dimensional solution space. Within this solution space, many different mappings are possible. We particularly emphasized the two approaches C and D which are derived by specifying constraints.

In principle, the essence of solution A is to use as many devices as possible. The solution can be approximated by a dynamic scheduling approach based on distributing work packages to devices on request. However, optimality cannot be guaranteed by a runtime approach. Solution B basically results in a mapping which reduces the degree of parallelism, since the most energy-efficient devices are used only. This can even lead to serializing an application by using only one device out of the set of the most energy-efficient ones. Therefore in many cases, solution B will not meet the requirements of a programmer. A strengthening of the solution is required to restrict the "serializing" behavior. This is achieved for solutions C and D by introducing a time and energy constraint.

In this chapter, we analyze the Time/Energy solution space and particularly focus on feasible solutions - mapping decisions which provide the best trade-offs between time and energy for work distribution based on our workload partitioning approach. Within the scope of this work, we extend our algorithm to a multi-objective version that enumerates Pareto optimal solutions, i.e. economical solutions which represent good trade-offs and constitute the Pareto Frontier. We also investigate the feasibility of constructing an approximate Pareto Frontier from which one can view the nature of the curve that represents trade-off regions between time and energy and decide for the best work

distribution strategy. Finally, we propose methods to recommend reasonable solutions from the Pareto Frontier set in cases when the user does not provide any constraints for any of the objectives.

Contributions

The main contribution of this work is the analysis of the bi-dimensional solution space. Since solutions of interest are part of the Pareto Frontier, we compute the Pareto Frontier to get a better understanding of the multi-objective optimization problem and to be able to propose reasonable solutions balancing performance and energy costs. More precisely, the contributions of this work are as follows:

(a) analysis of the feasible region of the bi-dimensional solution space

(b) computing Pareto Frontier based on the point-by-point approach

(c) efficient approximation of Pareto Frontier based on closed-line segments

(d) recommending a reasonable trade-off solution between time and energy

(e) experimental evaluation of the algorithms and construction methods

Related Work

Minimizing energy and makespan is an instance of classical multi-objective optimization problems[MA04, Deb14, ST93]. We group the related work in three parts covering related work in homogeneous cluster systems, heterogeneous cluster systems, and other domains. The main difference to our work from most research efforts is that we are explicitly interested in *optimal solutions* for data-parallel HPC applications executed in heterogeneous platforms.

Homogeneous systems. Optimizing makespan and energy in single- and multi-node homogeneous computational clusters has been studied in many papers. Freeh et. al. [FLP$^+$07] propose an approach in lowering energy consumption by dynamically adjusting power-performance states in high-performance homogeneous cluster nodes. He et. al. [HLCL05] tackles the bi-objective optimization problem for job-shop scheduling in homogeneous systems, while Durillo et. al. [DNP13] deal with the trade-off between energy-efficiency and makespan for workflow scheduling in systems with CPUs. The main difference to our work is that we deal with heterogeneous clusters.

Heterogeneous systems. Pineau et. al. [PRV11] and Li et. al. [LW12] consider minimizing energy consumption by scheduling tasks to machines and ensuring completion of tasks within a runtime constraint in heterogeneous architectures. We consider the inverse problem, which is more intricate, as shown in [RMH16].

Li et. al. [LLQ09] propose a heuristic method based on the MinMin algorithm to minimize energy consumption in a heterogeneous system by dynamically controlling the power state of each cluster node. Wang et. al. [WR10] propose a CPU-GPU inter-processor work distribution approach in a machine using DVFS to minimize energy consumption. Our problem differs, since we base our scheduling decisions on energy measurements, rather than dynamically changing the power states of the machines to meet a specific objective.

Balaprakash et. al. [BTW14] focus on energy-makespan optimizations specifically for multi-processor systems and Intel Xeon Phi. Friese et. al. [FKM+13] use evolutionary algorithm NSGA-II for bi-objective optimizations in a system with two types of CPUs. However, our optimization model targets larger heterogeneous cluster with a much higher degree of heterogeneity, including non-conventional processors.

Young et. al. [YAB+13] use various heuristic scheduling algorithms to allocate resources in an energy-constrained system for tasks with individual deadlines. They focus on completing as many tasks as possible by not violating the hard deadlines of the tasks. Our approach is different since instead of focusing on executing as many tasks as possible by not violating individual task deadlines, our optimization problem requires that all the work packages must be computed without violating the energy constraint.

Closest work to ours has been recently published by Tarplee et. al. [TFM+16]. The paper focuses on bi-objective optimizations between makespan and energy in heterogeneous clusters for bag-of-tasks applications, whereas we are targeting data-parallel HPC applications. The main goal of their work is to compute a Pareto Frontier with approximate makespan-energy trade-offs, while we compute Pareto optimal solutions. The bisection algorithm generates optimal solutions for the whole set of trade-offs between makespan and energy, as well as for energy-constrained optimizations. In addition we provide an efficient approximate algorithm which is based on a set of line segments finding solutions close to the optimum, whereas their approach is based on weighted sum technique and convex filling algorithm.

Other domains. Similar research work in multi-objective optimizations has been done in other domains [YWM+01, NDPB13] or with different optimization objectives [DJSS07]. Yang et. al. [YWM+01] provide an energy-aware task-scheduling method for embedded systems, while Nesmachnow et. al. [NDPB13] deal with minimizing energy consumption in grid systems. Optimizing makespan and reliability for a set of heterogeneous machines is discussed in [DJSS07], with special emphasis on the probability of failure of such machines. The main difference from these research works is that we are targeting heterogeneous computational clusters.

8.1. The Solution Space of Special Interest

In the previous chapter, we have proven that for a given energy constraint E_c, we can find a minimal makespan by minimizing the energy function $M(\lambda)$. However, the minimal makespan solution found by the bisection algorithm constitutes a singular optimal packing solution with makespan λ and an energy consumption $M(\lambda)$. Using an incremental approach to energy constraint E_c from solution B to solution A, we can find a set of intermediate solutions. This set of solutions permit the exploration of the solution space.

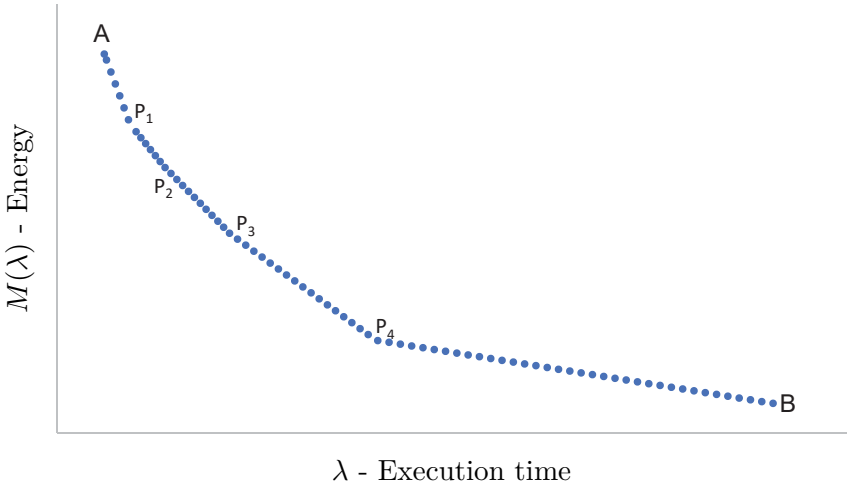


Figure 8.1.: Solution space for varying E_c

A typical solution space is shown in Figure 8.1[b]. Each of the singular solutions has been computed by Algorithm 4 introduced in Chapter 7. If each of these solutions is a Pareto optimal solution, i.e. for each of the points in the figure no other solution can be found which are better off in both dimensions, the solution space constitutes a Pareto Frontier (a set of all Pareto optimal solutions). Moreover, the solution space in Figure 8.1 reveals an interesting artifact: it has distinct line segments. This can be seen clearly in the figure at the breakpoints P_1 to P_4 where the slope of the curve changes. This raises the question if the curve is indeed piece-wise linear and if it can be constructed in a piece-wise fashion. We will tackle in detail these questions in the following sections.

[b]Figure 8.1 does not show all optimal solutions, only a selected representative sample in order to provide better visualization of the optimal solution space

8.1.1. Pareto Frontier

Computing the points of the Pareto frontier can be phrased as a multi-objective optimization minimizing makespan and energy simultaneously as the following mathematical program:

$$\begin{array}{lll} \text{minimize} & (\lambda, M(\lambda)) & \\ \text{subject to} & t_d \cdot x_d \leq \lambda & \forall d \in D \\ & \sum_{d \in D} x_d = n & \\ & x_d \in \mathbb{N}_0 & \forall d \in D \\ & \lambda \in \mathbb{R}_0^+ & \end{array}$$

We say that a solution of the above mathematical program with makespan and energy $(\lambda, M(\lambda))$ is *pareto optimal* if there is no other solution $(\lambda', M(\lambda'))$ such that $\lambda' < \lambda$ and $M(\lambda') < M(\lambda)$. This solution is defined as a *Strong Pareto Optimum* (SPO) for which there are no other solutions with smaller makespan or energy. A *Weak Pareto Optimum* (WPO) is a solution for which either the makespan or the energy can be improved, i.e. for a solution $(\lambda, M(\lambda))$ there is at least one solution where $\lambda' \leq \lambda$ and $M(\lambda') < M(\lambda)$ or $\lambda' < \lambda$ and $M(\lambda') \leq M(\lambda)$ [Jur08].

For a given system, the Pareto frontier or Pareto set is the set of parameterizations (allocations) that are all Pareto efficient (Pareto optimal). Finding Pareto frontiers is particularly useful in engineering. By yielding all of the potentially optimal solutions, a designer can make focused trade-offs within this constrained set of parameters, rather than needing to consider the full ranges of parameters.

Lemma 3. *Suppose that there exists a feasible schedule with energy at most E. Then, given energy parameter E, Algorithm 4 computes a strong Pareto optimal solution.*

Proof. Let λ^* be the minimum makespan possible among all feasible schedules with energy at most E. By construction, Algorithm 4 finds a schedule with makespan λ^* and energy $M(\lambda^*) \leq E$. Since $M(\lambda^*) \leq E$, the optimality of λ^* implies that there can be no schedule with makespan less than λ^* and energy at most $M(\lambda^*)$. Similarly, $M(\lambda^*)$ is the minimum energy possible among all feasible schedules with makespan at most λ^*, so there can be no schedule with energy less than $M(\lambda^*)$ and makespan at most λ^*. Therefore, the schedule found by Algorithm 4 (see Chapter 7) is a strong Pareto optimum. □

We prove that our algorithm computes strong Pareto optimal solutions. Unfortunately, the effort can be rather high since it requires iterative execution of the bisection algorithm to find all Pareto optimal solutions. Hence, in the next section, we analyze if there are alternative ways to approximate a Pareto Frontier faster.

8.1.2. Analysis of Pareto Optimal Mapping Decisions

Figure 8.2 depicts all Pareto optimal solutions for a given system, with points A and B representing two extreme solutions: most time-efficient solution $A(\lambda_A, M(\lambda_A))$, and most energy-efficient solution $B(\lambda_B, M(\lambda_B))$.

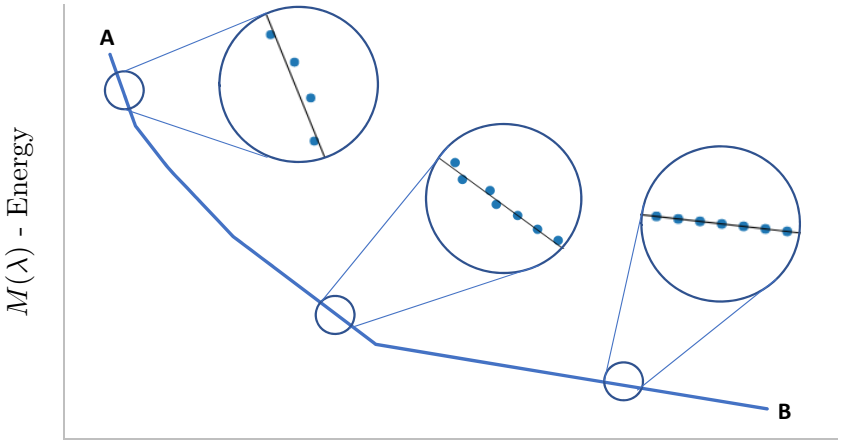


Figure 8.2.: Pareto Frontier

The Pareto Frontier in the figure leads us to some interesting observations:

Observation 1. Pareto Frontier is made up of a certain number of line segments with slightly different slopes. Our experimental data show that in each line segment, a different amount of devices has been used for mapping work packages, making the number of devices a property for mapping decisions.

Observation 2. Pareto solutions are non-collinear with the segments that can be drawn between breakpoints of the line segments. From the Figure 8.2 we can observe that apart from the first line segment from solution B towards solution A, in all other segments the Pareto optimal solutions are slightly offset from the line drawn between two breakpoints.

In this section, we will analyze the feasibility of finding the breakpoints of the Pareto Frontier, which would enable us to construct an approximate Pareto Frontier. We will tackle an important question in this context: Can we accurately construct a Pareto Frontier? These two observations will play a crucial role in our efforts to efficiently construct an approximate Pareto Frontier in a piece-wise linear fashion.

First, we analyze changes in the slope of the function when we add new devices in the computing process. Assume that the first device is the most

energy-efficient device and there is no other device with the same energy consumption. With that assumption the solution $(n \cdot t_1, n \cdot e_1)$ for n work packages is a Pareto optimal solution. It represents the most energy-efficient solution, which requires a single device. By increasing the energy constraint E_c of our bisection algorithm, we will obtain solutions with one or more devices by trading energy for a better makespan.

For each solution, we observe that k most energy-efficient devices are used to assign work packages with packing Algorithm 3 for $M(\lambda)$. Hence, a solution has an additional property, i.e., the number of devices required to produce this solution. Considering the monotonicity of the function $M(\lambda)$, with decreasing makespan, the number of devices used for constructing a solution will increase.

Let us analyze how these mapping solutions are generated. We have shown that Pareto optimal solutions lie in the domain interval between solution A and solution B. Assume we have four devices d_1, d_2, d_3, d_4 sorted from the most to the least energy-efficient. Their characteristics have been summarized in Table 8.1. Assume we have 9 work packages to assign.

Table 8.1.: Execution time and energy consumption for the devices

device	execution time per WP - t_d	energy consumption per WP - e_d
d_1	20	20
d_2	16	33
d_3	13	45
d_4	20	45

In Figure 8.3 we show the shifting process of work packages from one device to the other (using only two of the most energy-efficient devices) and the impact in makespan λ_X (left side) and total energy consumption $M(\lambda_X)$ (right side). Solution X in the figure corresponds to the solution B where device d_1, being the most energy-efficient, is processing all work packages and device d_2 none. Devices d_1, d_2 are sorted from the most energy-efficient to the least energy-efficient, such that energy consumption per work package is $e_1 \leq e_2$. We assumed that execution time per work package in the devices is $t_1 > t_2$, since otherwise solution B would coincide with solution A for all cases $t_1 \leq t_2$. In this case, achieving better performance requires shifting work packages from the first device to the second. We denote with $\lambda'_X, M(\lambda'_X)$ makespan and energy consumption after shifting the first work package to device d_2. As depicted with Solution X', makespan is reduced by $\Delta\lambda = |\lambda_X - \lambda'_X|$ or in this case equal to the execution time per one work package $\Delta\lambda = t_1$. On the other hand total energy consumption is increased by $\Delta M(\lambda) = |M(\lambda_X) - M(\lambda'_X)|$, or the difference in energy consumption between the two devices $\Delta M(\lambda) = |e_1 - e_2|$. The shifting process continues until the makespan cannot be lowered further, i.e. the accumulated execution times of the devices are balanced. This leads to the

last mapping solution for two devices X''''''. In Figure 8.3 we have omitted the shifting process for intermediate solutions $X'' - X''''$ for the sake of readability.

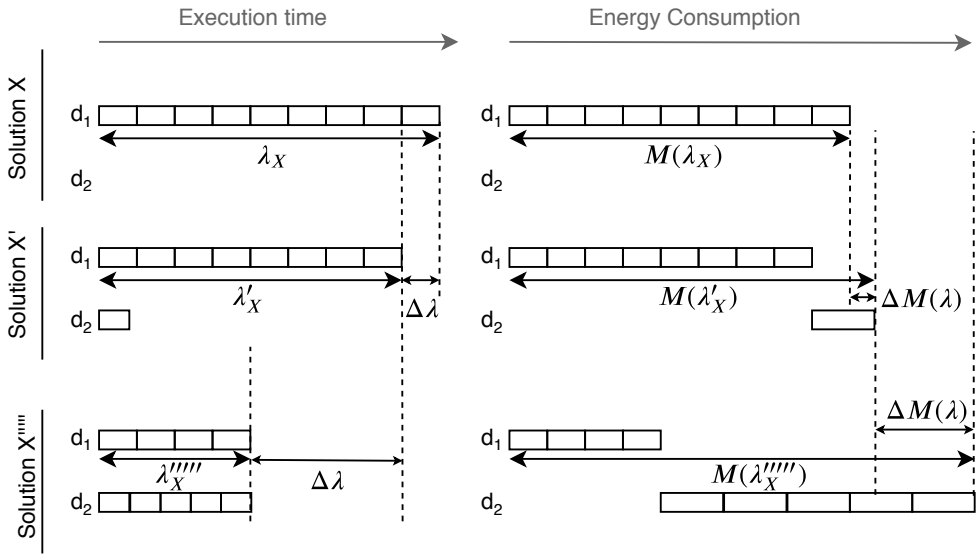


Figure 8.3.: Change in makespan and energy consumption during the work package shifting process for two devices

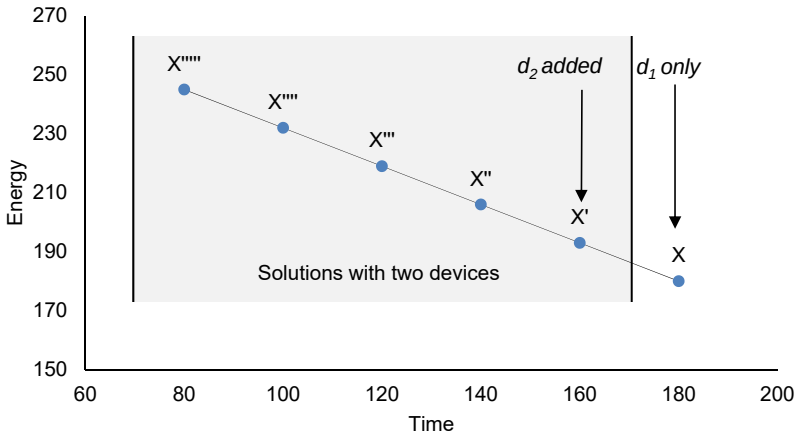


Figure 8.4.: Pareto curve - WP mapping solutions for two devices

As shown in Figure 8.4 each shift from solution X to X'''''' produces a Pareto optimal solution, since in a two-device setting there are no other mapping decisions for which one of the objectives can get better off without making the other objective worse. Also, as shown in the figure 8.4 we can observe that for a setting with two devices all Pareto optimal solutions are collinear with the line drawn between the endpoints of the segment respectively solutions X and X''''''.

This is due to the constant rates of change from one Pareto optimal solution to the next. For each shift, we have the same values for $\Delta\lambda$ and $\Delta M(\lambda)$, and the slope of the curve is determined by the ratio $\frac{\Delta M(\lambda)}{\Delta\lambda}$.

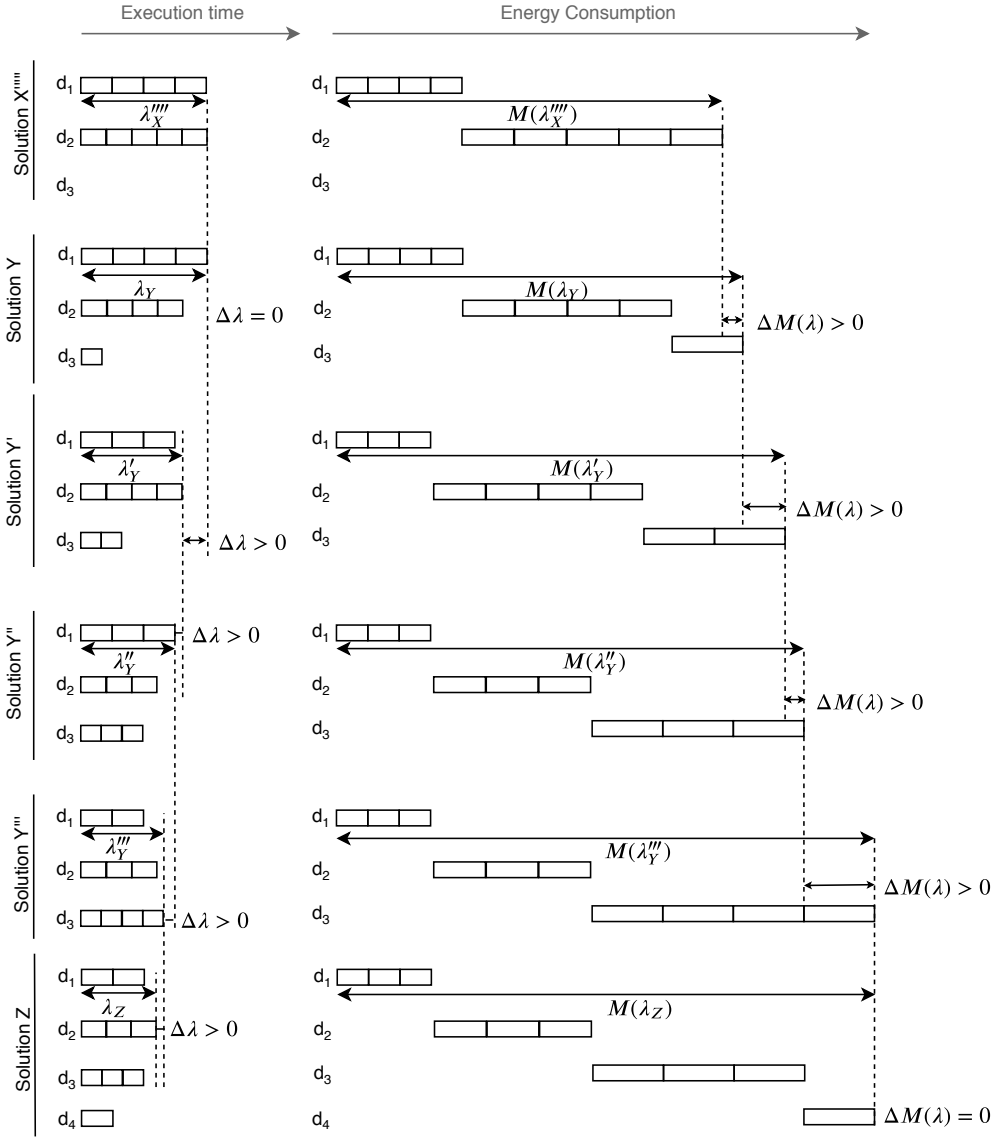


Figure 8.5.: Change in makespan and energy consumption during the work package shifting process for three and four devices

We expand our system to include devices d_3 and d_4 as shown in Figure 8.5. Devices are sorted according with the energy-efficiency key such as $e_1 \leq e_2 \leq e_3 \leq e_4$. Solution X'''' as depicted in the figure is the last solution which minimizes both objectives for a system with two devices. At this point,

the only way to decrease the makespan is to include a third device in the computing process. However, unlike in the settings with two devices, as shown with solution Y, a single shift does not necessarily decrease the makespan, since the makespan is equal to the maximal individual makespans among the devices. Hence, Solution Y is not a strong Pareto optimal solution since the makespan $\lambda_Y = \lambda''''_X$, while $M(\lambda_Y) > M(\lambda''''_X)$ - making Solution Y a weak Pareto optimal solution.

The next shift in order to decrease the makespan would be to shift one work package from d_1 to d_3. In this case Solution Y' is a strong Pareto optimal solution since for the new makespan value there are no shifting decisions which would result in smaller energy consumption. Makespan decrease between two Pareto optimal solutions is now $\Delta\lambda = |\lambda''''_X - \lambda'_Y|, t_1 \leq \Delta\lambda \leq t_2$ while energy consumption $\Delta M(\lambda) = |M(\lambda''''_X) - M(\lambda'_Y)| = |(e_1 - e_3)| + |(e_2 - e_3)|$. Solution Y'' is also a Pareto optimal solution, while Solution Y''' is the last mapping decision for the setting with three devices. As long as there are only three devices Solution Y''' is a strong Pareto optimal solution. However, if a fourth device is added and it has identical energy consumption figures as the third device, i.e. $e_3 = e_4$, a shift of one work package to the device d_4 from device d_3 would result in a decrease of makespan, without changes in energy consumption. In our case Solution Y''' is a weak Pareto optimal solution since Solution Z is superior.

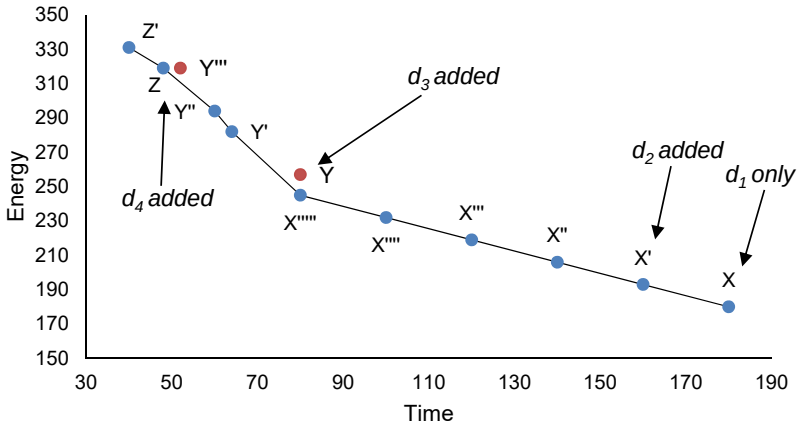


Figure 8.6.: Pareto curve - WP mapping solutions for three and four devices

In Figure 8.6 we have plotted mapping decisions discussed above. Solutions denoted with blue circles are strong Pareto optimal solutions, while weak Pareto optimal solutions are denoted with red circles. From the figure, we can see that Solutions Y and Y''' are weak Pareto optimal solutions. Solution Y is inferior to Solutions X'''' in terms of energy consumption, while Solution Y''' is inferior to Solution Z in terms of makespan. Solution Z' is the last mapping decision for this setting.

As shown above, generating a Pareto Frontier only by shifting work packages from one device to another is difficult, even if the devices are sorted by energy-efficiency and the shifting process takes into account the makespan reduction. Also, we have shown that the breakpoints of the Pareto curve do not always coincide with the first or last mapping decision for a specific number of devices. However, these mapping decisions are always in the vicinity of a strong Pareto optimal solution which defines the slope of the next line segment. Therefore, one can construct an approximate Pareto Frontier by extrapolating last mapping decisions at each step before adding a new device in computing. An approximate λ for the last mapping decisions before each device inclusion can be calculated by solving systems of equations which express balanced accumulated execution times between devices.

First, we assume that accumulated execution times are perfectly balanced at these points:

$$\begin{aligned} x_1 \cdot t_1 &= x_2 \cdot t_2 \\ &\dots \\ x_1 \cdot t_1 &= x_k \cdot t_k \\ &\dots \\ x_1 \cdot t_1 &= x_m \cdot t_m \end{aligned} \tag{8.1}$$

with the condition that all of the n work packages have to be assigned to m devices

$$x_1 + \dots + x_k + \dots + x_m = n \tag{8.2}$$

For each of the settings with $s, 0 < s \leq m$ devices, we substitute equations 8.1 in the equation 8.2. We get the mapping decisions for all devices for each of the s settings such that for the k-th device $0 < k \leq s$ of the setting s we have

$$x_k^s = \frac{n}{\sum_{i=1}^{s} \frac{t_k}{t_i}} \tag{8.3}$$

or the approximate λ for a setting with s devices:

$$\lambda_s^{\approx} = \max_{k=1..s} \frac{n \cdot t_k}{\sum_{i=1}^{s} \frac{t_k}{t_i}} \tag{8.4}$$

We say approximate λ since the relaxation above in 8.1 which assumes that the accumulated execution times are perfectly balanced may skew λ when accumulated times are not perfectly balanced. Having λ from the construction method 8.4 for each setting we can use Algorithm 2 to get the mappings for the most energy-efficient solution under a time constraint and calculate the energy consumption at these points. We can draw lines between these points and construct a curve which approximates Pareto Frontier.

Finally, we will address the second observation from the beginning of this

section, which deals with non-collinearity of the Pareto optimal solutions with lines drawn between the breakpoints of the Pareto Frontier. This jitter happens due to the variable rates of change in increase/decrease of makespan and energy consumption. An exception to this are settings with two devices. We noted above that in the case shown in Figure 8.3, Pareto optimal solutions are collinear with the line between two breakpoints as a result of a constant rate of change, i.e. $\Delta\lambda$ and $\Delta M(\lambda)$ increase or decrease by the same amount on each shift. Hence, $\frac{\Delta M(\lambda)}{\Delta\lambda} = const$ for all shifts, except when the last shift decreases λ less due to the requirements for balancing accumulated execution times. Further, as we increase the number of devices to more than two, the rate of change does not remain constant as we have observed in Figure 8.5. We see there that the values of $\Delta\lambda$ and $\Delta M(\lambda)$ are different for different work package shifts, making the ratio $\frac{\Delta M(\lambda)}{\Delta\lambda}$ respectively the slopes between each two neighboring Pareto optimal solutions variable. This variability causes all Pareto optimal solutions to be slightly offset from the line drawn between two breakpoints of the Pareto curve, even if the breakpoints coincide with Pareto optimal solutions.

As demonstrated above, one cannot compute a Pareto Frontier accurately in a piece-wise linear fashion considering the uncertainty of the breakpoints and the jitter in the approximated curve. However, construction of an approximate Pareto curve which would allow us to view the nature of the Pareto Frontier and see the trade-off regions between time and energy is feasible as long as it requires less effort and computes faster. In section 8.3 we will compare the runtime of Algorithm 4 for generating a Pareto Frontier and construction method 8.4 for approximating it. We will also measure and discuss the accuracy of the approximated Pareto curve.

8.2. Recommending Reasonable Trade-Off Solutions

Before coming to experiments, we want to tackle the problem of recommending trade-off solutions if the user does not specify any constraints. A trade-off between execution time and energy consumption is required which leads to a new problem: Is it possible to determine automatically without programmer support "good" solutions which represent reasonable compromises between execution time and energy consumption.

A first idea would be to recommend solutions which are best trade-offs and lie at the knee of the Pareto Curve. In Figure 8.7 we show three different cases where Pareto optimal solutions (blue circles) are distributed in a balanced way between time and energy (left); are more sensitive to towards energy consumption such that a mapping with smaller makespan results in energy consumption spike (middle); and more sensitive towards runtime (right). We have denoted with U and marked it with a red circle the Utopian solution which we define as a virtual mapping which combines time-efficiency as well as energy-efficiency. In this way, Pareto optimal solutions that are closest to the Utopian solutions are

best trade-offs between time and energy for a given system. The solution that has the smallest distance from the Utopian solution can be recommended as the best trade-off. This applies to all cases and takes into account the sensitivity towards time or energy for particular settings and applications.

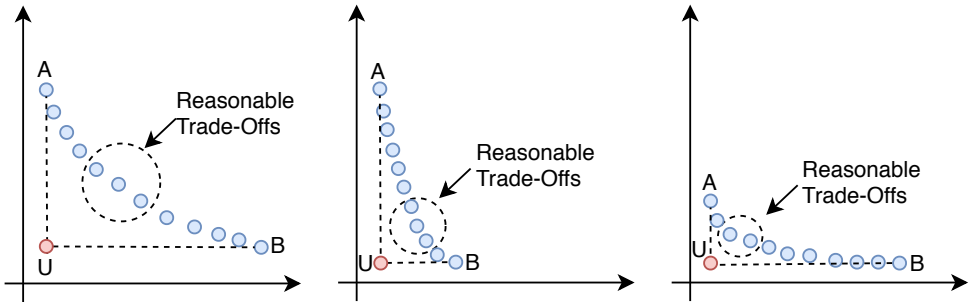


Figure 8.7.: Reasonable Trade-Off Solutions

For scale-invariant recommendation which selects a solution without taking into account the sensitivity of the Pareto curve towards time or energy, one can use the Weighted Euclidean Distance (WED) such that

$$d_{xu} = \sqrt{w_e(e_x - e_u)^2 + w_t(t_x - t_u)^2} \tag{8.5}$$

where d_{xu} denotes the Weighted Euclidean Distance from a solution x with components (t_x, e_x) to the utopian point u with components (t_u, e_u). Weights are set equal to the reciprocal of the variance of energy, respectively, execution time in the set of solutions. This leads to a special case of the Mahalanobis distance and implies that one can effectively transform both measurements on the same scale. The advantage of such an approach is that one can avoid meaningless effects (as simple Euclidean distance is not scale-invariant) due to an arbitrary choice of units for time and energy that are not directly comparable anyway.

8.3. Experimental Evaluation

In this section, we present the results from our experiments for runtime efficiency of the Algorithm 4 and construction method 8.4 to produce a Pareto Frontier. We also discuss the accuracy of the approximated Pareto curve by evaluating the error of the approximated Pareto Frontier towards Pareto optimal solutions. We experiment with different system configurations with 2, 3, 5 and 10 devices.

In this experiment we apply Algorithm 4 iteratively by decreasing the energy constraint to generate all Pareto optimal solutions and we use the construction method 8.4 and Algorithm 2 to find the breakpoints of the Pareto curve and

Table 8.2.: Performance of Algorithms and Error of Approximation

m	n	**Alg. Speedup** BIS/BPA	**Mean Error Range: 0 - 10**	**Maximal Error Range: 0 - 10**	**Referenced Figure**
2	10^3	56.5×	$6.38 \cdot 10^{-4}$	$1.28 \cdot 10^{-3}$	8.8
3	10^3	141.14×	$1.02 \cdot 10^{-3}$	$9.50 \cdot 10^{-3}$	8.9
5	10^3	139.61×	$6.90 \cdot 10^{-4}$	$4.50 \cdot 10^{-3}$	8.10
10	10^3	119.74×	$2.28 \cdot 10^{-3}$	$1.38 \cdot 10^{-2}$	8.11
10	10^4	1572.15×	$6.70 \cdot 10^{-5}$	$3.9 \cdot 10^{-4}$	-
10	10^6	206910.46×	$1.60 \cdot 10^{-6}$	$1.4 \cdot 10^{-5}$	-

construct an approximate Pareto Frontier.

We measure the time required to generate the Pareto Frontier with iterative application of bisection algorithm BIS, and the time required to construct an approximate Pareto Frontier with the construction method 8.4 - the breakpoints approach BPA.

In Table 8.2 we have summarized the results of the experiments. We show there the speedup of the construction method BPA in approximating a Pareto curve over BIS which generates a full set of optimal Pareto points. We observe that as the number of work packages increases, approximation of the Pareto Frontier is orders of magnitude faster than generation of the Pareto Frontier by enumerating all Pareto optimal solutions. BPA approximation is influenced only by the number of the devices and since in practice the number of work packages exceeds the number of devices, approximating is much faster, such as in the case of a setting with 10 devices and one million work packages - approximation of the Pareto curve is done more than 200.000 times faster. We presented in this table also the mean error and maximal error for each of the experiments. We measure error as the Euclidean distance of a Pareto optimal solution from the closest point on the approximated line. It appears as if with the increasing problem size the error value gets smaller, but this comes as a side-effect of normalization of the makespan and energy consumption values in the range $0 - 10$. We will discuss in more detail below the errors and accuracy of the approximation.

In Figure 8.8 we present the Pareto Frontier for a system with $m = 2$ devices and $n = 1000$ work packages. Execution time and energy consumption values have been normalized to a range between 0 and 10. Two blue points represent the breakpoints of the Pareto curve found by the construction method BPA, respectively the most energy-efficient and most time-efficient solution for a system with two devices. The blue line segment between the two points depicts the approximated Pareto curve. With orange triangles, we have denoted the Pareto optimal solutions produced by BIS algorithm. Due to a large number

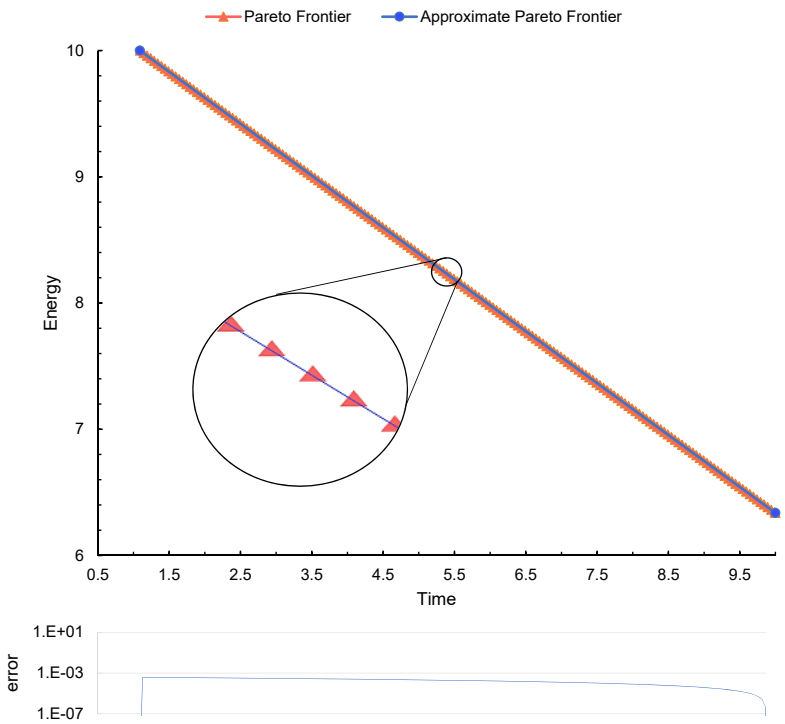


Figure 8.8.: Pareto Frontier for a system with two devices

of mapping decisions which are Pareto optimal, these markers appear as superimposed on each other, forming a thick orange line segment. However, zooming in into any part of the line shows us that these solutions are apart from each other. In the zoomed-in circle, we see Pareto optimal solutions overlapping with the line segment between two breakpoints.

In the bottommost part of the figure, we show the error of the approximated Pareto curve for each of the Pareto optimal solutions. The horizontal axis of the error chart matches with the horizontal axis of the Pareto chart. Hence for each of the Pareto solutions, we show the error of the approximated curve, respectively the Euclidean distance between the Pareto solution and the closest point on the approximated line. For this experiment, we observe that BPA has found both breakpoints, which coincide with Pareto optimal solutions. This is backed up by error value, which is zero for both points. The intermediate solutions have a small error value with mean error $6.38 \cdot 10^{-4}$ and the maximal error $1.28 \cdot 10^{-3}$ which is observed for the second Pareto optimal solution from the left of the chart. In a two-device setting where two breakpoints are found accurately, we intuitively expect the error of the approximation to be zero since as we discussed in previous sections the rate of change in such settings must be constant. However, in this case, we observe a non-zero error. This comes as a result of a different rate of change in the slope for the last Pareto optimal

solution. Up to the balancing of the execution times between two devices (prior to the last shift of a work package from the first device to the second), the makespan is determined by the first device (most energy-efficient device) since it has most of the work packages. For each shift, the makespan is decreased by an amount equal to the execution time per work package for the first device. However, the last shift may be a tipping point such that the makespan becomes determined by the second device and the decrease in makespan is less than the execution time per work package of the first device. This causes the rate of change $\frac{\Delta M(\lambda)}{\Delta \lambda}$ to be smaller than in all other prior shifts where the rate of change was constant. Since this point coincides with a breakpoint, it will skew the slope of the approximated line, introducing an error for all Pareto optimal points except the two extremes which coincide with the breakpoints. The error is bigger for the Pareto optimal solutions that are nearest to the last Pareto optimal solution (the leftmost point) and decreases to zero on the other end - the first Pareto optimal solution (the rightmost point). This happens only in cases when the makespan for the last Pareto optimal solution becomes determined by the second device, while in cases when makespan remains determined by the first device, the error is zero for all Pareto optimal solutions in a two-device setting.

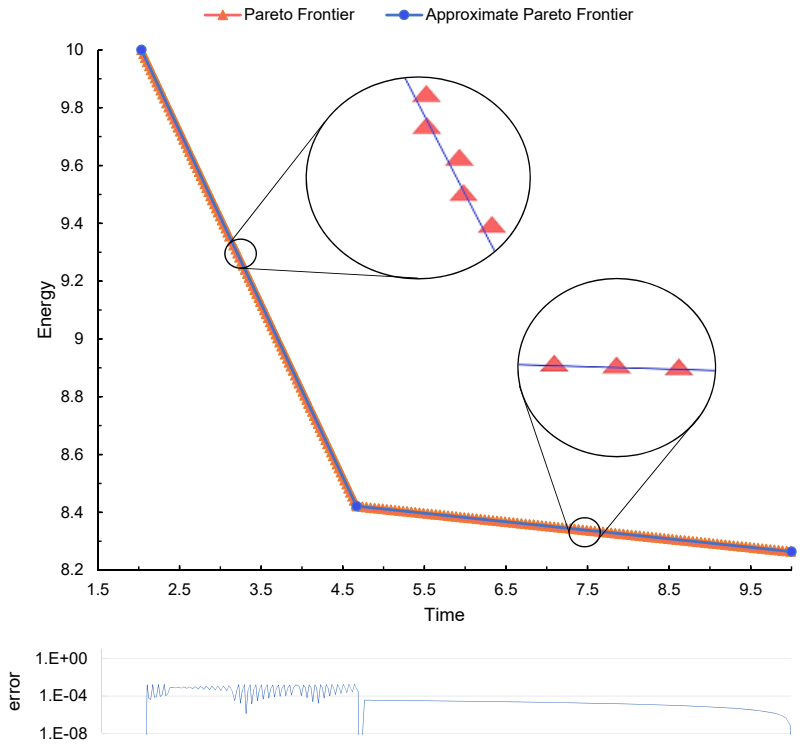


Figure 8.9.: Pareto Frontier for a system with three devices

We present the experiment with $m = 3$ devices and $n = 1000$ work packages in Figure 8.9.

We observe that the Pareto Frontier is not a straight line anymore and becomes curvier as we add more devices. Three breakpoints of the curve marked with three blue circles and produced by BPA coincide with Pareto optimal solutions as we can see from the error chart where error drops to zero. In the zoomed-in circles, we can see that optimal solutions in the right line segment are collinear with the approximated line and we observe a similar error rate for that part as in the experiment with two devices. Here also the makespan has been determined by the second device on the last shift. On the left steeper line segment, which consists of mapping decisions with three devices, we see variable error values for different points. This is an expected result considering the variability of the rates of changes in slope for mapping decisions with more than two devices, as we discussed in the previous sections.

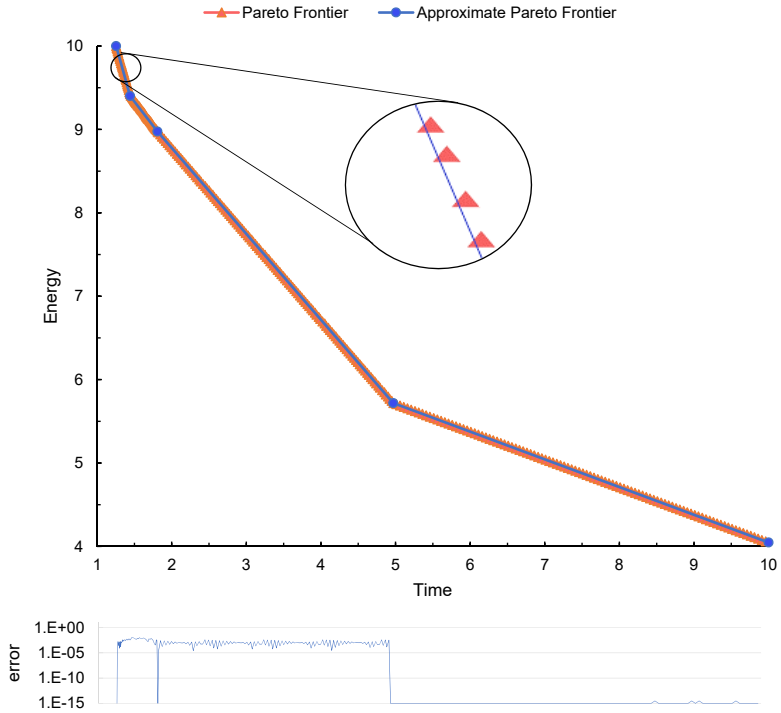


Figure 8.10.: Pareto Frontier for a system with five devices

In a similar fashion is approximated the Pareto Frontier for five devices (see Figure 8.10). In this case, BPA has found three breakpoints which coincide with strong Pareto optimal solutions as we can observe from the error chart, while two other breakpoints are weak Pareto optimal solutions. It is worth noting that in these cases the error value is similar to the error stemming from the collinearity of Pareto solutions with an approximated line between

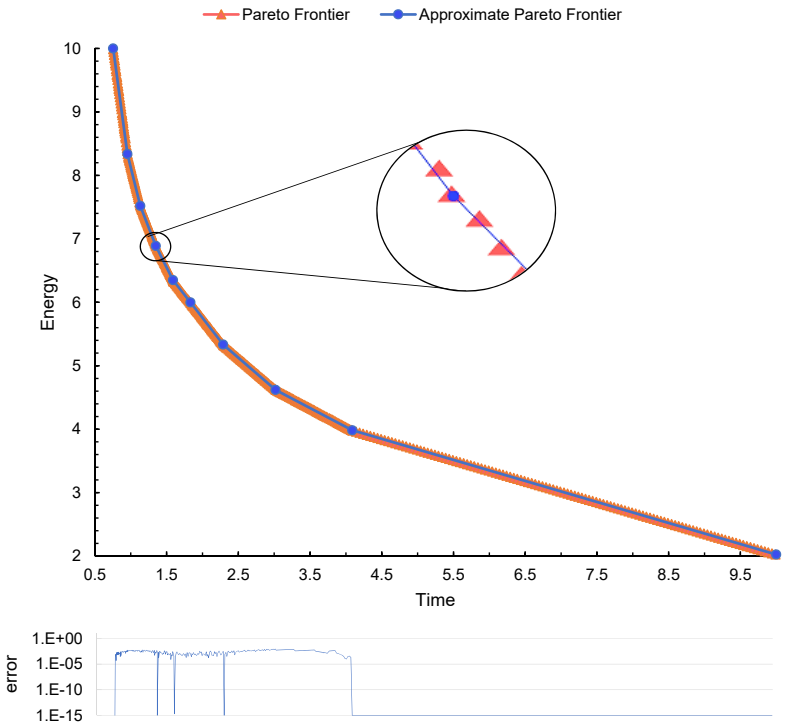


Figure 8.11.: Pareto Frontier for a system with ten devices

two strong Pareto optimums. Further, in this case, we observe that in the line segment consisting of mapping decisions for two devices only, the makespan on the last shift remains determined by the first device. Therefore, the error value is zero for all solutions on that segment.

Finally, we present the Pareto Frontier for a setting with 10 devices in Figure 8.11. As depicted, Pareto Frontier appears almost like a smoothed curve. *BPA* has found 10 breakpoints, 6 of which are strong Pareto optimums, one of them depicted in the zoomed-in circle superimposed on a Pareto optimal solution. Other breakpoints are weak Pareto optimums in the immediate vicinity of the strong Pareto optimums. We see similar variable error values for all settings with more than two devices.

Above, we have shown different cases of approximation of the Pareto Frontier, and in all cases, we have observed that approximation is much faster and the approximated line is very accurate with small error values that are in most cases negligible.

8.4. Summary

In this chapter we analyzed the bi-dimensional Time/Energy solution space for work distribution. We focused on a special region of the solution space, Pareto Frontier, which consists of mapping decisions that result in best trade-offs between makespan and energy consumption. We have proven that with bisection and packing algorithm introduced in the previous chapter we can find strong Pareto optimal solutions and by executing them iteratively we can generate a full set of Pareto optimal solutions - a Pareto Frontier. However, this process can be time-consuming therefore we put our effort in constructing an approximate Pareto Frontier by exploiting a property of the Pareto curve where the slope of the curve changes sharply when new devices are added. Our experimental results confirmed that approximating Pareto Frontier using this technique is much faster and the approximation error very small. A method to recommend reasonable trade-off solutions has also been discussed in this chapter. This method of recommendation is based on the Euclidean distance and selects a solution closest to a virtually superior point intersecting two extreme solutions - the most time-efficient and most energy-efficient solution.

Part IV.

Conclusions

CHAPTER 9

Conclusion

We conclude this thesis by summarizing our contributions and discussing possible direction of future research work.

9.1. Summary and Conclusions

In the course of work for this thesis, we have addressed some of the challenges posed in the context of High Performance Computing in asymmetric heterogeneous hardware architectures. We have identified and discussed various problems in this context and introduced approaches and techniques which enable efficient execution both in terms of execution time and energy consumption for HPC applications in heterogeneous clusters.

In this thesis, we have presented a workload partitioning approach based on work packages which expedites the partitioning and distribution process. By acknowledging the shortcomings and complexity of custom-size coarse-grain workload partitioning methods for heterogeneous platforms, we have proposed a partitioning approach which partitions data into same-size medium-grain data chunks. Partitioned data are encapsulated into work packages which include instructions for routing and execution process as well as profiling information. The main advantage of this approach is the simplification of the partitioning process and easier handling of device failures and poor performance. Further, it enables developing work distribution strategies to optimize for energy-efficiency besides runtime efficiency.

To alleviate programmer burden dealing with multi-paradigm programming models in heterogeneous systems, we have implemented a framework that combines MPI, OpenMP and OpenCL and provides an abstraction layer to the programmer. The framework supports workload partitioning and distribution in heterogeneous clusters and coordinates execution while ensuring workload balance between cluster nodes. To minimize the impact of data transfer in

the execution process, we have introduced a prefetching mechanism which runs in the cluster nodes and fetches work packages from front-end while devices are busy executing. By recognizing the likelihood of node and device failures during the execution process, we have implemented a failure handling procedure which reroutes work packages to other hardware resources and augments checkpoint-rollback-recovery fault-tolerance technique.

We have extended our programming approach to allow handling of more complex HPC applications. We analyzed various application characteristics and discussed challenges that may arise from intra-kernel and inter-kernel dependencies. We discussed the implications of dependencies and computational behavior of applications in partitioning strategies. As a result, we implemented a synchronization mechanism that deals with dependencies and ensures data coherency at all times during the execution process.

We discussed work distribution strategies based on our workload partitioning approach, which take into account energy efficiency as an additional component of the execution performance. We developed a set of algorithms which produce optimal solutions, respectively work package to device mapping decisions that minimize execution times or energy consumption. Two of these algorithms, A and B, produce optimal solutions that coincide with extremes in the time/energy solutions space. The former assigns work packages to devices to ensure the most time-efficient solution while the latter serializes execution to obtain the most energy-efficient execution. We introduced constraints on both optimization objectives, leading to algorithms C and D which produce optimal solutions by minimizing one of the objectives while upholding the constraint in the other.

We have further shown that by using bisection and packing algorithm iteratively, we can minimize both optimization objectives at the same time and generate a full set of optimal mapping decisions - the Pareto Frontier. Out of this set of Pareto optimal solutions, we can recommend a reasonable trade-off solution which combines runtime and energy-efficiency. Moreover, we analyzed the prospects of approximating the Pareto Frontier, which allows faster exploration of trade-off regions in order to speed up decision-making process on work distribution strategies.

9.2. Future Work

In this section, we discuss the potential directions of future research work stemming from different parts of this thesis.

Avoiding Bottlenecks in Data Transfers. Our work distribution approach has been designed to support medium-size heterogeneous clusters. The front-end is tasked with the transfer of the work packages to cluster nodes and receiving back processed work packages. We have seen from the experimental results that the prefetching mechanism running in cluster nodes minimizes transfer costs and their impact on the execution process. However, we recognize

the limitations of our approach for big size clusters and supercomputers. When the number of cluster nodes increases, the latency of the front-end responding to the requests of the prefetchers in cluster nodes also increases diminishing the scalability. A potential improvement would be to create multiple supervisors at different cluster nodes acting as front-end by serving a limited number of cluster nodes. Supervisors would need to coordinate to ensure data coherency.

Augmenting clusterCL with Kernel Parameter Tuning Techniques. Design and implementation of clusterCL are based on the idea of accelerating execution from the distribution of work between many devices within a heterogeneous system. Complementary to our approach, there are research works [FPB17], [NC15] which are focused on harvesting performance from heterogeneous devices by tuning various kernel parameters and ensure performance portability. Expanding clusterCL to include these techniques may lead to an increase in performance by lowering execution times of work packages in different devices.

Bibliography

[AMD] AMD. AMD Accelerated Parallel Processing SDK. `https://developer.amd.com/tools-and-sdks/`. [Online; accessed May-2019].

[AMW13] Naser M. Asghari, Michel Mandjes, and Anwar Walid. Optimizing energy management in multi-core servers. *SIGMETRICS Perform. Eval. Rev.*, 41(2):38–40, August 2013.

[AONM11] Ryo Aoki, Shuichi Oikawa, Takashi Nakamura, and Satoshi Miki. Hybrid OpenCL: Enhancing OpenCL for Distributed Processing. In *International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 149–154, May 2011.

[ARPS13] Albano Alves, Jose Rufino, Antonio Pina, and LuisPaulo Santos. clOpenCL - Supporting Distributed Heterogeneous Computing in HPC Clusters. In *Euro-Par 2012: Parallel Processing Workshops*, volume 7640 of *LNCS*, pages 112–122. Springer Berlin Heidelberg, 2013.

[Aug11] Augonnet, Cedric and Thibault, Samuel and Namyst, Raymond and Wacrenier, Pierre-Andre. Starpu: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurr. Comput. : Pract. Exper.*, 23(2):187–198, February 2011.

[BBD+19] Olivier Beaumont, Brett A. Becker, Ashley DeFlumere, Lionel Eyraud-Dubois, Thomas Lambert, and Alexey Lastovetsky. Recent advances in matrix partitioning for parallel computing on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 30(1):218–229, Jan 2019.

[BBNLS10] Amnon Barak, Tal Ben-Nun, Ely Levy, and Amnon Shiloh. A package for OpenCL based heterogeneous computing on clusters with many GPU devices. In *2010 IEEE Conference on Cluster Computing Workshops and Posters*, pages 1–7, Sept 2010.

[BBRR02] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. Partitioning a square into rectangles: Np-completeness

and approximation algorithms. *Algorithmica*, 34(3):217–239, Nov 2002.

[BF11] Richard L. Burden and J. Douglas Faires. *Numerical Analysis*. Brooks Cole Pub., USA, 2011.

[BTW14] Prasanna Balaprakash, Ananta Tiwari, and Stefan M. Wild. Multi Objective Optimization of HPC Kernels for Performance, Power, and Energy. In *High Performance Computing Systems. Performance Modeling, Benchmarking and Simulation*, volume 8551 of *LNCS*, pages 239–260. Springer International Publishing, 2014.

[Bun06] David P. Bunde. Power-aware scheduling for makespan and flow. In *Proceedings of the Eighteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 190–196, Cambridge, MA, 2006.

[CDDE13] Jesus Carabaño, Francisco Dios, Masoud Daneshtalab, and Masoumeh Ebrahimi. An exploration of heterogeneous systems. In *2013 8th International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pages 1–7, July 2013.

[Con19] OpenMPI Consortium. Open MPI: Open Source High Performance Computing. `http://mvapich.cse.ohio-state.edu/`, May 2019. [Online; accessed May-2019].

[Dal06] John T. Daly. A higher order estimate of the optimum checkpoint interval for restart dumps. *Future Generation Computer Systems*, 22(3):303 – 312, 2006.

[Deb14] Kalyanmoy Deb. *Multi-objective Optimization*, pages 403–449. Springer US, Boston, MA, 2014.

[DGAJ13] Tahir Diop, Steven Gurfinkel, Jason Anderson, and Natalie Enright Jerger. Distcl: A framework for the distributed execution of opencl kernels. In *2013 IEEE 21st International Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 556–566, Aug 2013.

[DJSS07] Jack J. Dongarra, Emmanuel Jeannot, Erik Saule, and Zhiao Shi. Bi-objective scheduling algorithms for optimizing makespan and reliability on heterogeneous systems. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 280–288, New York, NY, USA, 2007. ACM.

[DNP13] Juan José Durillo, Vlad Nae, and Radu Prodan. Multi-objective workflow scheduling: An analysis of the energy efficiency and makespan tradeoff. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, pages 203–210, May 2013.

[DPS+10] José Duato, Antonio J. Peña, Federico Silla, Rafael Mayo, and Enrique S. Quintana-Ortí. rCUDA: Reducing the number of GPU-based accelerators in high performance clusters. In *HPCS*, pages 224–231, June 2010.

[EA12] Barış Eskikaya and D. Turgay Altilar. Distributed OpenCL distributing OpenCL platform on network scale. *Int'l Journal of Computer Applications, vol. ACCTHPCA*, ACCTHPCA(2):25–30, 2012.

[ECLV12] Maja Etinski, Julita Corbalán, Jesús Labarta, and Mateo Valero. Parallel job scheduling for power constrained HPC systems. *Parallel Computing*, 38(12):615 – 630, 2012.

[FKM+13] Ryan Friese, Bhavesh Khemka, Anthony A. Maciejewski, Howard Jay Siegel, Gregory A. Koenig, Sarah Powers, Marcia Hilton, Jendra Rambharos, Gene Okonski, and Stephen W. Poole. An analysis framework for investigating the trade-offs between system performance and energy consumption in a heterogeneous computing environment. In *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum, Cambridge, MA, USA, May 20-24, 2013*, pages 19–30, 2013.

[FLP+07] Vincent W. Freeh, David K. Lowenthal, Feng Pan, Nandini Kappiah, Rob Springer, Barry L. Rountree, and Mark E. Femal. Analyzing the energy-time trade-off in high-performance computing applications. *IEEE Trans. Parallel Distrib. Syst.*, 18(6):835–848, June 2007.

[Fly72] Michael J. Flynn. Some computer organizations and their effectiveness. *IEEE Transactions on Computers*, C-21(9):948–960, Sep. 1972.

[FPB17] Jiří Filipovič, Filip Petrovič, and Siegfried Benkner. Autotuning of opencl kernels with global optimizations. In *Proceedings of the 1st Workshop on AutotuniNg and aDaptivity AppRoaches for Energy Efficient HPC Systems*, ANDARE '17, pages 2:1–2:6, New York, NY, USA, 2017. ACM.

[fS09] International Organization for Standardization. Portable Operating System Interface (POSIX®) Base Specifications. Standard, ISO, Geneva, CH, March 2009.

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A comprehensive performance comparison of cuda and opencl. In *2011 International Conference on Parallel Processing*, pages 216–225, Sep. 2011.

[GDF14] Philipp Gschwandtner, Juan J. Durillo, and Thomas Fahringer. Multi-Objective Auto-Tuning with Insieme: Optimization and Trade-Off Analysis for Time, Energy and Resource Usage. In *Euro-Par 2014 Parallel Processing*, volume 8632 of *LNCS*, pages 87–98. Springer International Publishing, 2014.

[GHK+12] Benedict Gaster, Lee Howes, David Kaeli, Perhaad Mistry, and Dana Schaa. *Heterogeneous Computing with OpenCL*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2012.

[GO11] Dominik Grewe and Michael F. P. O'Boyle. A static task partitioning approach for heterogeneous systems using opencl. In Jens Knoop, editor, *Compiler Construction*, pages 286–305, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[GPCF13] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer. LibWater: Heterogeneous Distributed Computing Made Easy. In *ICS*, pages 161–172, Eugene, Oregon, 2013.

[GPCF14] Ivan Grasso, Simone Pellegrini, Biagio Cosenza, and Thomas Fahringer. A uniform approach for programming distributed heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 74(12):3228 – 3239, 2014. Domain-Specific Languages and High-Level Frameworks for High-Performance Computing.

[GTO13] Andrey V. Gorobets, Francesc Xavier Trias, and Assensi Oliva. A parallel mpi+openmp+opencl algorithm for hybrid supercomputations of incompressible flows. *"Computers And Fluids"*, 88:764 – 772, 2013.

[GXS+12] Scott Grauer-Gray, Lifan Xu, Robert Searles, Sudhee Ayalasomayajula, and John Cavazos. Auto-tuning a high-level language targeted to gpu codes. In *2012 Innovative Parallel Computing (InPar)*, pages 1–10, May 2012.

[HLCL05] Yan He, Fei Liu, Hua-jun Cao, and Cong-bo Li. A bi-objective model for job-shop scheduling problem to minimize both energy

consumption and makespan. *Journal of Central South University of Technology*, 12(2):167–171, 2005.

[HS87] Dorit S. Hochbaum and David B. Shmoys. Using dual approximation algorithms for scheduling problems theoretical and practical results. *J. ACM*, 34(1):144–162, 1987.

[HSdSM10] Tobias Hilbrich, Martin Schulz, Bronis R. de Supinski, and Matthias S. Müller. Must: A scalable approach to runtime error detection in mpi programs. In Matthias S. Müller, Michael M. Resch, Alexander Schulz, and Wolfgang E. Nagel, editors, *Tools for High Performance Computing 2009*, pages 53–66, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Int17] Intel. Cilk Plus. `https://www.cilkplus.org/`, March 2017. [Online; accessed May-2019].

[Int19] Intel. Thread Building Blocks. `https://www.threadingbuildingblocks.org/`, May 2019. [Online; accessed May-2019].

[Jur08] Jurgen Branke, Kalyanmoy Deb, Kaisa Miettinen, Roman Slowinski. *Multiobjective Optimization: Interactive and Evolutionary Approaches*. Springer Science & Business Media, Germany, 2008.

[KH12] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., CA, 2nd edition, 2012.

[KHD+17] Guangyuan Kan, Xiaoyan He, Liuqian Ding, Jiren Li, Ke Liang, and Yang Hong. A heterogeneous computing accelerated SCE-UA global optimization method using OpenMP, OpenCL, CUDA, and OpenACC. *Water Science and Technology*, 76(7):1640–1651, 06 2017.

[Khr14] Khronos OpenCL Working Group. The OpenCL C Specification. Version 2.0. `http://www.khronos.org/opencl`, November 2014. [Online; accessed May-2019].

[KKLL11] Jungwon Kim, Honggyu Kim, Joo Hwan Lee, and Jaejin Lee. Achieving a Single Compute Device Image in OpenCL for Multiple GPUs. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 277–288, San Antonio, TX, USA, 2011.

[KSA+10] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating performance and portability of opencl programs. In *Proc. Automatic Performance tuning*, 2010.

[KSG12] Philipp Kegel, Michel Steuwer, and Sergei Gorlatch. dOpenCL: Towards a Uniform Programming Approach for Distributed Heterogeneous Multi-/Many-Core Systems. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, pages 174–186, May 2012.

[KSL+12] Jungwon Kim, Sangmin Seo, Jun Lee, Jeongho Nah, Gangwon Jo, and Jaejin Lee. SnuCL: An OpenCL Framework for Heterogeneous CPU/GPU Clusters. In *Proceedings of the 26th ACM International Conference on Supercomputing (ICS)*, pages 341–352, San Servolo Island, Venice, Italy, 2012.

[KT06] Jon M. Kleinberg and Éva Tardos. *Algorithm design*. Addison-Wesley, 2006.

[LFC13] Thibaut Lutz, Christian Fensch, and Murray Cole. PARTANS: An Autotuning Framework for Stencil Computation on multi-GPU Systems. *ACM Trans. Archit. Code Optim.*, 9(4):59:1–59:24, January 2013.

[LHK09] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55, Dec 2009.

[Li12] Keqin Li. Optimal configuration of a multicore server processor for managing the power and performance tradeoff. *J. Supercomput.*, 61(1):189–214, July 2012.

[Lib19] Lawrence Livermore National Library. Message Passing Interface (MPI). `https://computing.llnl.gov/tutorials/mpi/`, May 2019. [Online; accessed May-2019].

[LL12] Qiang Liu and Wayne Luk. Heterogeneous systems for energy efficient scientific computing. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 7199 of *LNCS*, pages 64–75. Springer Berlin Heidelberg, 2012.

[LLA+13] Palden Lama, Yan Li, Ashwin M. Aji, Pavan Balaji, James Dinan, Shucai Xiao, Yunquan Zhang, Wu-chun Feng, Rajeev Thakur, and Xiaobo Zhou. pVOCL: Power-Aware Dynamic Placement and

Migration in Virtualized GPU Environments. In *Proceedings of ICDCS*, pages 145–154, Philadelphia, PA, 2013.

[LLQ09] Yu Li, Yi Liu, and Depei Qian. A heuristic energy-aware scheduling algorithm for heterogeneous clusters. In *2009 15th International Conference on Parallel and Distributed Systems*, pages 407–413, Dec 2009.

[LPB18] Tiago Lobato Gimenes, Flavia Pisani, and Edson Borin. Evaluating the performance and cost of accelerating seismic processing with cuda, opencl, openacc, and openmp. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 399–408, May 2018.

[LR06] Alexey Lastovetsky and Ravi Reddy. HeteroMPI: Towards a message-passing library for heterogeneous networks of computers. *Journal of Parallel and Distributed Computing*, 66(2):197 – 220, 2006.

[LSM15] Janghaeng Lee, Mehrzad Samadi, and Scott Mahlke. Orchestrating multiple data-parallel kernels on multiple devices. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*, pages 355–366, Oct 2015.

[LSPM13] Janghaeng Lee, Mehrzad Samadi, Yongjun Park, and Scott Mahlke. Transparent CPU-GPU Collaboration for Data-parallel Kernels on Heterogeneous Systems. In *Proceedings of the 22Nd International Conference on Parallel Architectures and Compilation Techniques*, PACT '13, pages 245–256, Piscataway, NJ, USA, 2013. IEEE Press.

[LW12] Dawei Li and Jie Wu. Energy-aware scheduling for frame-based tasks on heterogeneous multiprocessor platforms. In *2012 41st International Conference on Parallel Processing*, pages 430–439, Sept 2012.

[MA04] R. Timothy Marler and Jasbir S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and Multidisciplinary Optimization*, 26(6):369–395, Apr 2004.

[MGM+11] Aaftab Munshi, Benedict Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison-Wesley Professional, 1st edition, 2011.

[MPI12] MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.0. `http://www.mpi-forum.org`, September 2012. [Online; accessed May-2019].

[NC15] Cedric Nugteren and Valeriu Codreanu. Cltune: A generic auto-tuner for opencl kernels. In *Proceedings of the 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip*, MCSOC '15, pages 195–202, Washington, DC, USA, 2015. IEEE Computer Society.

[NDPB13] Sergio Nesmachnow, Bernabé Dorronsoro, Johnatan E. Pecero, and Pascal Bouvry. Energy-aware scheduling on multicore heterogeneous grid computing systems. *Journal of Grid Computing*, 11(4):653–680, Dec 2013.

[nVi19a] nVidia. CUDA Zone. `https://developer.nvidia.com/cuda-zone`, May 2019. [Online; accessed May-2019].

[nVi19b] nVidia. OpenCL SDK. `https://developer.nvidia.com/opencl`, March 2019. [Online; accessed May-2019].

[Ope19a] OpenACC. OpenACC Standard. `https://www.openacc.org/`, 2019. [Online; accessed May-2019].

[Ope19b] OpenMP. OpenMP Specification. `https://www.openmp.org/`, 2019. [Online; accessed May-2019].

[OWG13] Michael F. P. O'Boyle, Zheng Wang, and Dominik Grewe. Portable mapping of data parallel programs to opencl for heterogeneous systems. In *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, CGO '13, pages 1–10, Washington, DC, USA, 2013. IEEE Computer Society.

[PBAL13] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Self-adaptive ompss tasks in heterogeneous environments. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 138–149, May 2013.

[PRV11] Jean-François Pineau, Yves Robert, and Frédéric Vivien. Energy-aware scheduling of bag of tasks applications on master worker platforms. *Concurrency and Computation: Practice and Experience*, 23(2):145–157, 2011.

[RBW+19] Ari Rasch, Julian Bigge, Martin Wrodarczyk, Richard Schulze, and Sergei Gorlatch. docal: high-level distributed programming with opencl and cuda. *The Journal of Supercomputing*, Mar 2019.

[RM15] Valon Raca and Eduard Mehofer. Device-Sensitive Framework for Handling Heterogeneous Asymmetric Clusters Efficiently. In *26th IEEE International Symposium on Computer Architecture and High Performance Computing*, pages 181–188, Florianopolis, Brazil, Oct 2015.

[RMH16] Valon Raca, Eduard Mehofer, and Marcus Hudec. Optimal time and energy efficient work distributions in heterogeneous systems. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP)*, pages 440–447, Heraklion, Greece, Feb 2016.

[RRB+08] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA. In *PPoPP '08*, pages 73–82, Salt Lake City, UT, 2008.

[RW15] Rafal Rózycki and Jan Weglarz. Solving a power-aware scheduling problem by grouping jobs with the same processing characteristic. *Discrete Appl. Math.*, 182(C):150–161, February 2015.

[SCL+12] Ching-Lung Su, Po-Yu Chen, Chun-Chieh Lan, Long-Sheng Huang, and Kuo-Hsuan Wu. Overview and comparison of opencl and cuda technology for gpgpu. In *2012 IEEE Asia Pacific Conference on Circuits and Systems*, pages 448–451, Dec 2012.

[SFSV12] Jie Shen, Jianbin Fang, Henk Sips, and Ana Lucia Varbanescu. Performance gaps between openmp and opencl for multi-core cpus. In *2012 41st International Conference on Parallel Processing Workshops*, pages 116–125, Sep. 2012.

[SKG11] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. Skelcl - a portable skeleton library for high-level gpu programming. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1176–1182, May 2011.

[SPB16] Hercules Cardoso Da Silva, Flavia Pisani, and Edson Borin. A comparative study of sycl, opencl, and openmp. In *2016 International Symposium on Computer Architecture and High Performance Computing Workshops (SBAC-PADW)*, pages 61–66, Oct 2016.

[ST93] David B. Shmoys and Éva Tardos. Scheduling unrelated machines with costs. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 448–454, Philadelphia, PA, USA, 1993.

[SVL+16] Jie Shen, Ana Lucia Varbanescu, Yutong Lu, Peng Zou, and Henk Sips. Workload partitioning for accelerating applications on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 27(9):2766–2780, Sep. 2016.

[SVMS15] Jie Shen, Ana Lucia Varbanescu, Xavier Martorell, and Henk Sips. Matchmaking applications and partitioning strategies for efficient execution on heterogeneous platforms. In *2015 44th International Conference on Parallel Processing*, pages 560–569, Sep. 2015.

[TFM+16] Kyle M. Tarplee, Ryan Friese, Anthony A. Maciejewski, Howard Jay Siegel, and Edwin K. P. Chong. Energy and makespan tradeoffs in heterogeneous computing systems using efficient linear programming techniques. *IEEE Transactions on Parallel and Distributed Systems*, 27(6):1633–1646, June 2016.

[TFMS15] Kyle M. Tarplee, Ryan Friese, Anthony A. Maciejewski, and Howard Jay Siegel. Efficient and scalable pareto front generation for energy and makespan in heterogeneous computing systems. In *Recent Advances in Computational Optimization*, volume 580 of *Studies in Computational Intelligence*, pages 161–180. Springer International Publishing, 2015.

[TG07] Rajeev Thakur and William Gropp. Test suite for evaluating performance of mpi implementations that support mpi_thread_multiple. In Franck Cappello, Thomas Herault, and Jack Dongarra, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 46–55, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[TKS+11] Peter Thoman, Klaus Kofler, Heiko Studt, John Thomson, and Thomas Fahringer. Automatic opencl device characterization: Guiding optimized kernel design. In Emmanuel Jeannot, Raymond Namyst, and Jean Roman, editors, *Euro-Par 2011 Parallel Processing*, pages 438–452, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[Top18] Top500. Top 500 List. `https://www.top500.org/lists/2018/11/`, 2018. [Online; accessed May-2019].

[Uni19] The Ohio State University. MVAPICH software. `http://mvapich.cse.ohio-state.edu/`, May 2019. [Online; accessed May-2019].

[WR10] Guibin Wang and Xiaoguang Ren. Power-Efficient Work Distribution Method for CPU-GPU Heterogeneous System. In *Symposium on Parallel and Distributed Processing with Applications (ISPA)*, pages 122–129, Taipei, Taiwan, Sept 2010.

[WWO14] Yuan Wen, Zheng Wang, and Michael F. P. O'Boyle. Smart multi-task scheduling for opencl programs on cpu/gpu heterogeneous platforms. In *2014 21st International Conference on High Performance Computing (HiPC)*, pages 1–10, Dec 2014.

[XBZ+12] Shucai Xiao, P. Balaji, Qian Zhu, R. Thakur, S. Coghlan, Heshan Lin, Gaojin Wen, Jue Hong, and Wu chun Feng. VOCL: An optimized environment for transparent virtualization of graphics processing units. In *InPar*, pages 1–12, May 2012.

[YAB+13] B. Dalton Young, Jonathan Apodaca, Luis Diego Briceño, Jay Smith, Sudeep Pasricha, Anthony A. Maciejewski, Howard Jay Siegel, Bhavesh Khemka, Shirish Bahirat, Adrian Ramirez, and Yong Zou. Deadline and energy constrained dynamic resource allocation in a heterogeneous computing environment. *The Journal of Supercomputing*, 63(2):326–347, Feb 2013.

[You74] John W. Young. A first order approximation to the optimum checkpoint interval. *Commun. ACM*, 17(9):530–531, September 1974.

[YWM+01] Peng Yang, Chung Wong, P. Marchal, F. Catthoor, D. Desmet, D. Verkest, and R. Lauwereins. Energy-aware runtime scheduling for embedded-multiprocessor socs. *IEEE Design Test of Computers*, 18(5):46–58, Sept 2001.