Novel Data-Partitioning Algorithms for Performance and Energy Optimization of Data-Parallel Applications on Modern Heterogeneous HPC Platforms

Hamidreza Khaleghzadeh

UCD student number: 15209602

The thesis is submitted to University College Dublin
in fulfilment of the requirements for the degree of
Doctor of Philosophy in Computer Science

School of Computer Science and Informatics

Head of School: Professor Pádraig Cunningham

Research Supervisor: Assoc. Prof. Alexey Lastovetsky

March 2019

Acknowledgements

First and foremost, I would like to thank my supervisor Dr Alexey Lastovetsky for accepting me to do my PhD into the Heterogeneous Computing Laboratory (HCL), and for his immense guidance kept me on the right path during my study. I am greatly indebted to him for his valuable advice, and encouragement.

I would like to extend my special gratitude to Dr Ravi Reddy Manumachu for his guidance and encouragement through the duration of my study. To all my colleagues in UCD's Heterogeneous Computing Laboratory for fruitful collaborations: Semen Khokhriakov, Ken O'Brien, Muhammad Fahad, Emin Nuriyev, Arsalan Shahid and Tania Malik. Especial thanks to Elayne Ruane for her help proofreading the thesis.

The last three years of my life have been truly rewarding for becoming a member of the highly prestigious University College Dublin. From the UCD School of Computer Science, thank you to Dr Pádraig Cunningham, head of school, both Dr Michela Bertolotto and Dr Tahar Kechandi, my DSP members. Thank you also to the school's support staff: Lorraine McHugh, D'Arcey Jackson, Paul Martin and Tony O'Gara for their consistent helpfulness over the years.

This research has emanated from research conducted with the financial support of Science Foundation Ireland (SFI) under Grant Number 14/IA/2474. I would like to thank SFI and University College Dublin for their financial support in the form of scholarship awards. I am also grateful to the financial support of COST Action IC0805 "Open European Network for High-Performance Computing on Complex Environment" for the interesting schools and workshops held in University of Calabria (Italy), University College Dublin (Ireland) and Ruđer Bošković Institute (Croatia).

I would like to thank the many amazing friends I met in Ireland who made

this journey full of pleasures. I also extend my appreciation to Mr David and Ms Marie Redmond for their support, affection, and a home which they provided me during my PhD education.

A heartfelt thanks to my parents and sister Homa. I am grateful for the encouragement, support and love throughout my whole life.

Last but not least, I am thankful to God for all his blessings in my entire life.

To my family.

Abstract

Heterogeneity has turned into one of the most profound and challenging characteristics of today's HPC environments. Modern HPC platforms have become highly heterogeneous owing to the tight integration of multicore CPUs and accelerators (such as Graphics Processing Units, Intel Xeon Phis, or Field-Programmable Gate Arrays) empowering them to maximize the dominant objectives of performance and energy efficiency. Designed for legacy homogeneous platforms, traditional parallel algorithms and tools will deliver a small fraction of the potential performance and energy efficiency that we should expect from highly hybrid HPC platforms in the future.

Performance and energy are the two most dominant objectives for optimization on modern heterogeneous HPC platforms such as supercomputers and cloud computing infrastructures. Recent research on modern homogeneous multicore platforms demonstrates that the performance and energy profiles of data-parallel applications executing on such platforms exhibit drastic variations due to inherent complexities in these platforms such as severe contention for shared resources and Non-Uniform Memory Access (NUMA).

In this thesis, we present that these inherent characteristics and complexities have posed serious challenges to modelling and optimization of data-parallel applications on modern heterogeneous platforms for performance and energy. We illustrate that the discrete functional relationships between perfor-

mance and workload size and between energy and workload size have non-linear and non-convex shapes, which deviate significantly from the shapes and assumptions that allowed state-of-the-art optimization algorithms to find optimal solutions for performance and energy consumption. Thereby we demonstrate that the *workload distribution* has become an important decision variable that can no longer be ignored on modern heterogeneous HPC platforms.

We formulate the problem of optimization of data-parallel applications on modern heterogeneous HPC platforms for performance and dynamic energy and then propose two new model-based data partitioning algorithms, which are named *HPOPTA* and *HEOPTA*. These algorithms respectively minimize the execution time and the dynamic energy consumption of computations in the parallel execution of applications. We also present two other algorithms, *HEPOPTA* and *HTPOPTA*, for solving bi-objective optimization problems for execution time and dynamic energy, and also execution time and total energy on modern heterogeneous HPC platforms, respectively. All these algorithms consider one decision variable, *workload distribution*. Unlike traditional approaches looking for load-balanced solutions, solutions returned by the algorithms are, generally speaking, non-balanced.

In a typical hybrid node, the tight integration of accelerators with multicore CPUs via PCI-E communication links contains inherent limitations such as limited main memory of accelerators and limited bandwidth of the PCI-E communication links. These limitations pose formidable programming challenges to the execution of *large workload sizes* on these accelerators. In this research, we describe an out-of-card library, which is called *HCLOOC*, containing interfaces that address these challenges. It employs optimal software pipelines to overlap data transfers between host CPU and the accelerator and computations on the accelerator. It is designed using the fundamental building blocks,

which are OpenCL command queues for FPGAs, Intel offload streams for Intel Xeon Phis, and CUDA streams and events that allow concurrent utilization of the copy and execution engines provided in Nvidia GPUs.

We experimentally analyse and demonstrate the optimality and efficiency of the proposed algorithms and library using two well-known scientific data-parallel applications, matrix multiplication and 2D fast Fourier transform, on a cluster of two highly heterogeneous nodes. Each application invokes highly optimized vendor specific kernels for CPUs and accelerators. The matrix multiplication application is implemented using *HCLOOC*, which allows the accelerators to run computations of any arbitrary workload size.

[1]

[1]This research has financially supported by Science Foundation Ireland (SFI) under Grant Number 14/IA/2474.

Contents

List of Figures

List of Tables

Statement of Original Authorship

I hereby certify that the submitted work is my own work, was completed while registered as a candidate for the degree stated on the Title Page, and I have not obtained a degree elsewhere on the basis of the research presented in this submitted work.

Chapter 1

Introduction

Due to the superior performance per watt of accelerators, modern High Performance Computing (HPC) platforms have become highly heterogeneous owing to the tight integration of multicore CPUs and accelerators (such as Graphics Processing Units, Intel Xeon Phis, or Field-Programmable Gate Arrays). Because of this inherent characteristic, processing elements contend for shared on-chip resources, such as Last Level Cache (LLC), interconnect, etc., and shared nodal resources, such as DRAM, Quick Path Interconnect (QPI), PCI-E links, etc. The severe resource contention and also Non-Uniform Memory Access (NUMA) have posed serious challenges to model and algorithm developers. Moreover, accelerators feature limited main memory compared to the multicore CPU hosts and are connected to the hosts via limited bandwidth PCI-E links thereby requiring support for efficient out-of-card execution.

Now, we study briefly the history of heterogeneous HPC platforms to discern how this most common architectural characteristic has emerged. For more than three decades prior to mid-2000s, computer users came to expect performance doubling every 18 months due to Moore's law and Dennard scaling [4, 5]. Both clock rate and power increased rapidly. This is the era of homogeneous and heterogeneous clusters of single-core processors. However, by 2004, computer designers hit the power wall caused by problems stemming from increasing power consumption and increasing power density (amount of power dissipated per unit area, which represents the heat dissipa-

tion). The power problem was caused primarily by the breakdown of Dennard scaling, a model whereby the power density of a transistor-based processor of a unit area remains constant due to voltage and current scaling down with the length of the transistor. Up until 2004, moving to a smaller transistor process meant frequency could be increased for no increase in heat dissipation. The breakdown of Dennard scaling meant frequency scaling was no longer economical [6]. The chip fabrication industry turned to multicore CPU architectures to address this problem of increased power consumption and power density. Frequency scaling was abandoned in favour of multiple processors per chip [7].

Around 2001, general purpose computing on GPUs became practical with the appearance of programmable shaders and floating point support. The release of CUDA by Nvidia in 2007 and subsequent availability of high-quality high-performance scientific libraries (CUBLAS, CUFFT, etc.) fuelled the extensive use of GPUs by the scientific community, especially for matrix computations. In addition to enhanced programmability, their superior energy efficiency (performance per watt or energy per flop) compared to multicore CPUs for certain class of HPC applications has been one of the prominent reasons behind their rapid adoption by the HPC community. Intel entered the accelerator market by launching their Xeon Phi coprocessors based on its Many Integrated Core (MIC) architecture in 2013. Its key selling point compared to programming on GPUs is the re-usability of existing parallel approaches (and reduced portability concerns) due to its standard x86 programming model. Therefore, because of their progressively improving programmability and better energy efficiency, accelerators have become an integral part of HPC platforms addressing the twin critical concerns of performance and energy consumption.

Figure 1.1 summarizes the percentage of system share for accelerators in Top500 Supercomputers [8] over a period of ten years between 2009 and 2018. According to the graph, there is a strong upward trend in the prevalence of accelerators during this time where the system share of accelerators increased from 1.4% in 2009 to about 28% in 2018.

The current Top500 list [8] includes 126 systems with Intel/AMD multi-core CPUs and Nvidia GPU accelerators and 31 systems with Intel Xeon Phi accel-

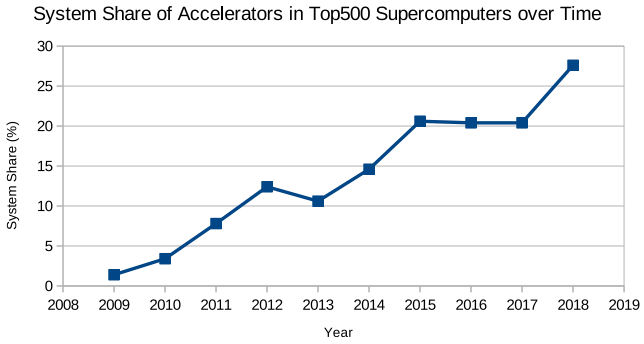

Figure 1.1: System share of accelerators in Top500 Supercomputers over a period of ten years between 2009 and 2018.

erators. Furthermore, there are four homogeneous clusters with hybrid nodes consisting of Intel Xeon Phi and Nvidia accelerators.

1.1 Motivations of This Research

Performance and energy have become the two most dominant objectives for optimization on modern heterogeneous HPC platforms such as supercomputers and cloud computing infrastructures [9, 10].

Heterogeneity and tight integration have introduced daunting challenges to the optimization of data-parallel applications for performance and energy consumption on modern HPC platforms. To explain the motivations of this research, we first elucidate unprecedented difficulties posed to performance and energy optimization of modern heterogeneous HPC platforms. Then, the necessity of a new model-based algorithm to address the bi-objective optimization problem for performance and energy on heterogeneous platforms is studied. Finally, we will explain how limited memory size on accelerators and the limited bandwidth of PCI-E communication links affect the execution of large problem sizes on these accelerators.

1.1.1 Shortcomings of State-of-the-art Load-balancing Algorithms for Performance Optimization on Modern Heterogeneous Platforms

Prior to presenting use cases that elucidate challenges for performance optimization on clusters of heterogeneous processors, we briefly study the evolution of performance models and data partitioning algorithms that have attempted to realistically capture the real-life behaviour of applications executing on these platforms for performance maximization.

The simplest technique is Constant Performance Model (CPM) which characterizes the speed of applications using positive constant numbers such as normalized processor speed, normalized cycle time, task computation time, average execution time, etc. [11, 12, 13]. The common aspect of these models is that the performance of a processor is assumed to have no dependence on the size of the workload.

The most advanced load-balancing algorithms use Functional Performance Model (FPM), which is application-specific. The FPMs represent the speed of a processor by a continuous function of problem size while satisfying some assumptions on its shape [14, 15]. The assumptions require them to be smooth enough in order to guarantee that optimal solutions minimizing the computation time are always load-balanced. The FPMs capture accurately the real-life behaviour of applications executing on nodes consisting of uniprocessors (single-core CPUs).

However, modern HPC platforms have complex nodal architectures with a highly hierarchical arrangement and tight integration of processors where resource contention and NUMA are inherent. On such platforms, the performance profiles of real-life scientific applications are not smooth and may deviate significantly from the shapes that allowed traditional and state-of-the-art load-balancing algorithms to find optimal solutions.

Lastovetsky et al. [16] study the drastic deviations in the performance profiles for a real-life scientific application, Multidimensional Positive Definite Advection Transport Algorithm (MPDATA), in Xeon Phi co-processors. MPDATA is a core component of the Eulerian/semi-Lagrangian fluid solver (EU-

LAG) geophysical model [17], which is an established computational model developed for simulating thermo-fluid flows across a wide range of scales and physical scenarios. The authors propose an optimization technique reusing an advanced performance model of computation (FPM) but using novel load distribution to minimize the computation time of the application. Lastovetsky et al. [3, 18] illustrate in depth these variations in performance and energy profiles of two widely known and highly optimized scientific routines, OpenBLAS Double-precision General Matrix Multiplication (DGEMM) [19] and FFTW [20], on a modern multicore Intel Haswell CPU platform. They explain the limitations of the FPM-based load-balancing algorithms proposed in [21, 22, 23, 24, 25, 26, 27, 28, 29]. They propose novel model-based methods and algorithms for minimization of time and dynamic energy of computations for the most general performance and energy profiles of data parallel applications executing on homogeneous multicore clusters. Unlike load-balancing algorithms, optimal solutions found by these algorithms may not load-balance an application.

Nevertheless, the model-based methods, which have been proposed in [16, 3, 18], cannot be used for performance optimization of data-parallel applications on HPC platforms with hybrid nodes since they are designed for *homogeneous* clusters, i.e., cluster of identical processors.

We now present one motivational use case to elucidates the additional challenges that arise in HPC platforms with heterogeneous nodes. We use a hybrid NUMA platform which contains an Intel Haswell multicore CPU consisting of 24 physical cores with 64 GB main memory and an Nvidia K40c GPU.

We study the performance profiles of a 2D Fast Fourier Transform (FFT) application, computing the 2D-FFT of $M \times 51200$ matrices. It executes a highly optimized native kernel for CPU and the accelerator. The structure of the application will be explained later in Chapter 3.

Figure 1.2 shows speed functions for the CPU and GPU. Each profile is a discrete function of performance against the problem size. One can observe significant fluctuations in the performance profile, which we call variations. The variation is related to the difference of speed between two subsequent local minima (s_1) and maxima (s_2), which is defined below:

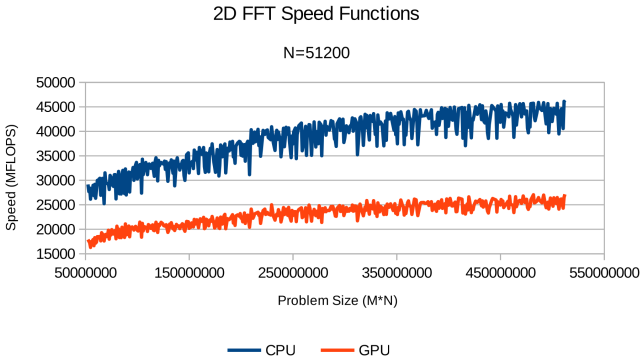

Figure 1.2: Speed functions of heterogeneous 2D FFT application executing on a heterogeneous node including an Intel multicore CPU and an Nvidia GPU.

$$variation(\%) = \frac{|s_1 - s_2|}{\min(s_1, s_2)} \times 100 \tag{1.1}$$

To make sure that our experimental results are reliable, and it is not noise that is the underlying cause behind these variations, the experiments for each data point in speed functions are repeated until sample means for execution times of all the two kernels running in parallel on the CPU and GPU fall in the interval with the confidence level 95 percent, and a precision of 0.1 (10%) is achieved. The statistical methodology is explained in detail in Appendix A. Briefly, the methodology contains the following main steps: 1) We make sure the platform is fully reserved and dedicated to our experiments and exhibits clean behaviour by monitoring its load continuously for a week. 2) For each data point in the speed functions of an application, the sample mean is used, which is calculated by executing the application repeatedly until the sample mean lies in the user-defined confidence interval, and the user-defined precision has been achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions by checking the density plots of the observations.

From the figure, the maximum variations for CPU and GPU are almost 33% and 20%, respectively. These shapes violate the assumptions on the shape of FPMs. Therefore, current load-balancing data partitioning algorithms based

on FPMs may not return optimal solutions.

The presented use case illustrates the dramatic variations observed in performance profiles of highly optimized scientific applications executing on heterogeneous HPC platforms. These variations are not singular and will become common because chip manufacturers are increasingly featuring tighter integration of processor cores, memory, and interconnect in their products. It is these variations that have now made the optimization problem for performance on such platforms difficult to solve. Moreover, the state-of-the-art load-balancing algorithms based on FPMs and even the model-based methods proposed in [16, 3, 18] for modern multicores are not equipped to deal with such cases where different processors exhibit different shapes of speed functions. In Chapter 3, we will show more speed functions including lots of variations and present a novel model-based data partitioning algorithms that employ a *load-imbalancing* parallel computing method to address the new challenges.

1.1.2 Shortcomings of State-of-the-art Energy Optimization Algorithms on Modern Heterogeneous Platforms

In this section, we first present the drastic changes observed in the shape of energy profiles of real-life scientific data-parallel applications running on heterogeneous HPC platforms compared to parallel platforms composed of uniprocessors. Then, the challenges posed to solving energy optimization problem by the new complexities in modern HPC platforms are highlighted using this presentation.

In the era of single and dual-core processors, energy profiles of real-life scientific applications were linear or smooth with minimal variations. Yang et al. [30] take advantage of these simple and uncomplicated profiles and propose an energy optimization algorithm by assuming that the energy consumption with higher workload is larger than that with lower workload. In [3], the authors experimentally illustrate that the shape of energy and performance profiles of applications running on processors with one or a few cores is almost smooth. They mathematically prove that balancing load for performance leads to energy optimization in such platforms. In general, principal features of popular

algorithms, proposed for optimization of energy consumption on these systems, are:

- Employing analytical modelling to estimate the dynamic energy consumption of applications [31, 32, 33, 30, 34, 35].
- Input parameters to these models (algorithms) are CPU/accelerator utilization, memory utilization, number of active threads, cache miss rate, bus traffic, CPU temperatures, etc. [32, 36, 37, 38, 39, 40, 41, 42, 43, 44]. Most of the existing methods do not consider application-level parameters for modelling.
- Apart from [3, 45, 18, 46], all methods assume a linear correlation between energy consumption and workload size and therefore do not consider workload distribution as a decision variable. Although considering variations, the aforementioned scientific efforts [3, 45, 18, 46] target homogeneous platforms for performance and energy optimization.
- Main decision variables in these methods are processor frequencies (adjusted using Dynamic Voltage and Frequency Scaling (DVFS)) and the number of cores/threads [31, 32, 33, 47, 48, 49].

Increasing the number of cores in a single die and also tight integrating of multi-core CPUs with many-core accelerators have incurred new complexities such as severe contention on shared resources (Last level caches (LLC), main memory, PCI-E links, etc.) and NUMA. Like performance optimization, these complexities lead to complicated nodal architectures, and henceforth, introduce new challenges to the optimization of data-parallel applications on these platforms for energy.

Lastovetsky and Reddy [3, 45, 18] examine energy profiles of parallel applications on *homogeneous* multicore CPUs. They demonstrate that energy profiles on modern platforms are not smooth because of severe resource contention, hierarchical arrangement and tight integration of processors. They experimentally show a complex and non-linear relationship between workload

size and dynamic energy consumption. They propose a model-based algorithm for optimization of dynamic energy consumption for data parallel applications executing on homogeneous clusters of multicore CPUs. The key input to the proposed algorithm is energy functions which model the real-life dynamic energy consumption of applications against problem size. Due to designing for homogeneous clusters, this algorithm cannot be used for minimizing the dynamic energy consumption of data-parallel applications on heterogeneous HPC platforms.

Heterogeneity and tight integration of multicore CPUs with accelerators cause new additional difficulties in energy optimization on modern hybrid HPC platforms. One visible manifestation of these complexities is a complex functional relationship between energy consumption and workload size of applications executing on these platforms where the shape of energy profiles may be highly non-linear and non-convex with drastic variations. We now elucidate these challenges with one use case.

Figure 1.3 shows dynamic energy profiles for the 2D fast Fourier transpose application running on the heterogeneous node which consists of an Intel Haswell multicore CPU, including 24 physical cores with 64 GB main memory, and an Nvidia K40c GPU. Each profile presents the dynamic energy consumption of a given processor versus problem size, running on the processor. In this experiment, The dynamic energy consumptions are measured using Watts Up Pro power meter. We will elaborate how these fine-grained dynamic energy profiles have been built via real measurement in Chapter 4.

From the energy function plot, one can observe:

- Energy variations for CPU and GPU are respectively around 90% and 60% for many workload sizes,
- Energy profiles of real-life scientific applications executing on modern HPC platforms are not smooth and may significantly deviate from the shapes observed on uni-processors,
- There is a complex correlation between dynamic energy consumption and workload size where workload distribution has now become an im-

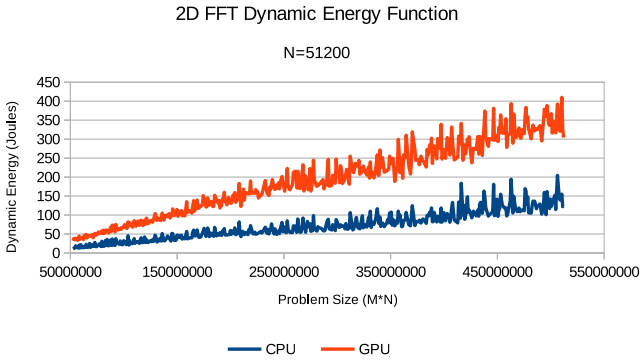

Figure 1.3: Dynamic energy functions of heterogeneous 2D FFT application executing on a heterogeneous node including an Intel multicore CPU and an Nvidia GPU.

portant decision variable that should not be ignored in solving energy optimization problem.

As explained earlier, except [3, 45, 18, 46] aiming to optimize performance and dynamic energy on homogeneous platforms, all proposed methods do not consider workload size as a parameter for energy modelling and even as a decision variable for energy optimization. They consider a linear relationship between workload size and dynamic energy consumption. However, regarding some scientific works [50, 51, 52, 3, 45] and the proposed use case, one can conclude that profiles on modern HPC platforms may be complex, which makes the relationship between workload size and dynamic energy consumption non-linear and even non-convex. Therefore, models which ignore workload size are not realistic and cannot reveal the exact behaviour of applications on modern heterogeneous platforms. Applying these approaches consequently leads to sub-optimal solutions for dynamic energy optimization problems.

The second challenge in energy optimization on modern hybrid HPC platforms is how to measure and model dynamic energy on these environments. While the execution time of every single computational kernel in a given hybrid application can be measured accurately using high precision timers (processor clocks), there is no such effective equivalent for measuring the dynamic

energy consumption. There are two dominant approaches to determine the energy consumption of a given application kernel: a). hardware based such as using on-chip power sensors or external power meters, b). software based such as energy predictive models.

Due to tight integration and severe contention, software-based approaches for modelling energy consumption on hybrid HPC platforms have been reported to be inaccurate [53, 52, 54, 55]. A vast majority of existing optimization algorithms for energy consumption rely on analytical modelling which have poor accuracy and high prediction error.

On the other hand, physical approaches provide accurate measurements. But they cannot determine intra-node energy consumption and fine-grained decomposition of the energy consumption during the application execution in a hybrid platform. Unlike homogeneous platforms, heterogeneous ones involve a wide diversity of processors where each has its unique profile with different dynamic energy consumption. That is, these approaches are not able to determine the amount of energy consumed by each kernel of a parallel application separately.

To summarize, there exist two main challenges to solving energy optimization problem on heterogeneous HPC platforms:

1. A model-based dynamic energy optimization algorithm which considers workload distribution as a decision variable,

2. A practical approach to accurately determine the decomposition of dynamic energy consumption for each kernel running on heterogeneous HPC platform using physical measurements.

Apart from dynamic energy consumption, the enormous total energy consumption in data centres and big clusters is also a critical constraint. The amount of static energy consumed by idle computers in clusters and clouds is non-negligible [56]. To save total energy consumption, the idle computers in clusters, clouds, web-servers and big data centers are switched off or put in sleep mode [57, 58, 9, 59, 60]. Now, there is an open question: “Does dynamic energy optimization always lead to total energy optimization?”. We are going to study this issue in this thesis.

1.1.3 Necessity of Novel Bi-objective Optimization Algorithms for Performance and Energy on Modern Heterogeneous Platforms

In the previous sections, we studied the performance and dynamic energy functions of a data-parallel application, 2D FFT, and explained that workload distribution has now become an important decision variable that should be taken into account in solving performance and also energy optimization problems on modern heterogeneous HPC platforms.

In this section, first, one use case is presented to highlight how performance optimization can impact dynamic energy consumption and vice versa. We then illustrate the challenges posed to solving bi-objective optimization problem for performance and dynamic energy on heterogeneous HPC platforms.

Figure 1.4 shows the globally optimal Pareto-front containing 16 solutions for an input problem size $n = 11184 \times 51200$ of the 2D FFT application, that its performance and dynamic energy functions are presented in Figures 1.2 and 1.3, respectively. We have considered workload distribution as the only decision variable. As shown in the figure, the workload distribution, which maximizes the performance, has an execution time of 1.29 seconds and a dynamic energy consumption of 219 joules. The workload distribution, with the minimal dynamic energy consumption of 151 joules, has an execution time of 1.72 seconds. From these results, one can conclude that optimizing for dynamic energy consumption alone degrades performance by 33%, and optimizing for performance alone increases dynamic energy consumption by 45%.

We can observe significant numbers of trade-off solutions for performance and dynamic energy when workload distribution is considered as the decision variable.

State-of-the-art solutions for bi-objective optimization problem for performance and energy on heterogeneous platforms can be broadly classified into *system-level* and *application-level* categories. System-level methods [61, 62, 63, 10, 64, 47] aim to optimize performance and energy of the system or the environment where the applications are executed. They employ

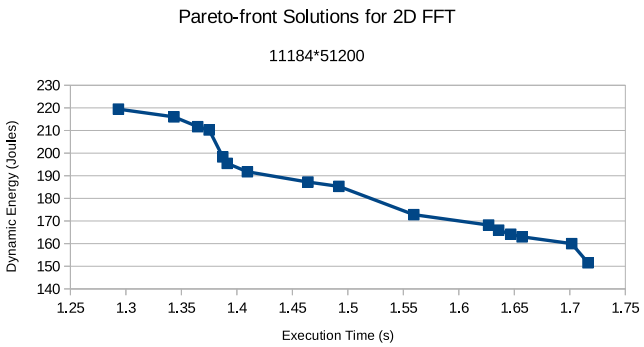

Figure 1.4: Pareto-front solutions of heterogeneous 2D FFT application for a given problem size 11184×51200 *running on a heterogeneous platform including one Intel multicore CPU and an Nvidia GPU.*

application-agnostic models and hardware parameters as decision variables where the dominant decision variable in this category is DVFS.

Application-level methods [65, 66, 67, 49, 68, 69, 70, 71, 72, 73, 50, 51] use application-level parameters and models for predicting the performance and the energy consumption of applications to solve bi-objective optimization problem for performance and energy. The key decision variables are the number of threads and the number of processors. Along with decision variables, several parameters, such as the cost of floating-point operations, cost of memory operations, latencies and bandwidths of the communication links, etc., are considered, which impact the performance and energy consumption of the applications but which have fixed values in the methods. These approaches also consider workload size as an application parameter but assume a linear relationship between performance and workload size and between energy consumption and workload size. Therefore, they do not consider workload distribution as a decision variable.

Reddy et al. [45] study the Bi-objective Optimization Problem for Performance and dynamic Energy (BOPPE) for data-parallel applications on homogeneous clusters of modern multicore CPUs. It employs only one decision variable, the workload distribution. They present an efficient and exact global optimization algorithm called *ALEPH* that solves the BOPPE. It takes as inputs, discrete functions of performance and dynamic energy consumption

against problem size, and returns the globally Pareto-optimal set of solutions. Extensions of the algorithm are proposed for employment in self-adaptable applications where low runtime overhead and low memory footprint are crucial [46]. It should be noted that the works in [45, 46] target *homogeneous* HPC platforms.

In summary, regarding non-linear profiles for performance and dynamic energy on heterogeneous HPC platforms and their complex relationship with workload size, we need a model-based data partitioning algorithm to solve BOPPE for data-parallel applications on these platforms. The algorithm should take into consideration the variations in the discrete profiles. To address this issue, in Chapter 5, we will propose a data partitioning algorithm which considers *load-imbalanced* solutions that are totally ignored by load-balancing approaches.

1.1.4 Challenges to Execution of Large Problem Sizes on Accelerators

Integration of multicore CPUs with accelerators poses challenges to execution of large workload sizes on these accelerators. These challenges, arising from the limited main memory of accelerators and their tight integration with multicore CPUs via PCI-E communication links, are listed below:

1. **Limited main memory of accelerators:** The size of main memory in accelerators is small compared to that of the host multicore CPU connected to it. Regarding the programming model for accelerators, all data, required by any kernel, should be loaded into the accelerator memory prior to any kernel invocation. Therefore, the maximum problem size that can be solved by an accelerator is limited by its main memory size.

2. **Limited bandwidth of the PCI-E communication link:** Kernel executions on accelerators usually entail multiple data transfers of data structures from the host CPU to the accelerator and back. Accelerators are connected to CPUs using PCI-E communication links. However,

due to the limited bandwidth of the PCI-E communication link, this impacts the execution times of applications. Accelerators such as GPUs provide advanced hardware support to facilitate overlap of data transfers between host and device and computations on the device. Therefore, libraries aiming to provide efficient implementations for accelerators must take into account the differences in hardware support for effective communication-computation overlap to optimize their software pipelines for kernel implementations.

3. **Shortage of libraries supporting large workload sizes on accelerators:** There is an abysmal lack of libraries providing interfaces that allow programmers to write implementations for their data-parallel kernels on accelerators which are able to execute large workload sizes.

According to the aforementioned limitations, we need an efficient out-of-card library which facilitates utilization of accelerators to solve big instances of data-parallel applications.

Generally, in this thesis, *out-of-card* algorithms are referred to as methods designed to process a workload which is too large to fit into an accelerator's main memory at one time. On the other hand, *out-of-core* computations use disk storage in case the problem is too large to fit into a computer's main memory.

1.2 Contributions of This Research

To summarize, our main contributions in this thesis are:

1. Studying the realistic and accurate behaviour of data parallel applications on modern heterogeneous clusters of hybrid nodes and the challenges introduced to model and algorithm design because of resource contention and NUMA for their performance and dynamic energy consumption optimization. We will demonstrate that workload distribution has now become an important decision variable that cannot be ignored in solving the performance and energy optimization problems.

2. An efficient data partitioning algorithm, which is named *HPOPTA*, for optimization of data-parallel applications on heterogeneous HPC platforms for performance. We will demonstrate that optimal solutions found by *HPOPTA* may not load-balance an application.

3. Demonstrating the efficiency of *HPOPTA* for large-scale simulated clusters and introducing a hierarchical two-level workload distribution algorithm using *HPOPTA* on a cluster of identical hybrid nodes.

4. A methodology to determine decomposition of dynamic energy consumption using physical measurements for heterogeneous hybrid servers with sufficient accuracy for energy optimization algorithms.

5. An efficient data partitioning algorithm, which is called *HEOPTA*, for optimization of data-parallel applications on heterogeneous HPC platforms for dynamic energy.

6. A model-based data partitioning algorithm, which is called *HEPOPTA*, for solving the bi-objective optimization problem for performance and dynamic energy for data-parallel applications on heterogeneous HPC platforms. We demonstrate that solutions provided by these algorithms do significantly improve the performance and energy efficiency of matrix multiplication and 2D fast Fourier transform in comparison with the traditional load-balanced configurations of the applications.

7. A model-based data partitioning algorithm, which is called *HTPOPTA*, for solving the bi-objective optimization problem for performance and total energy for data-parallel applications on heterogeneous HPC platforms. The efficiency of solutions found by *HTPOPTA* will be compared with algorithms only consider load-balancing distributions.

8. A library, which is called *HCLOOC*, that allows programmers to write out-of-card implementations of data-parallel kernels for accelerators such as GPUs, Xeon Phis, and FPGAs. The library is developed to address the challenges to validating the research in this thesis, arising from the limited main memory of accelerators. The library involves:

- An efficient out-of-card implementation written using *HCLOOC* of matrix multiplication for NVidia GPUs.
- An efficient out-of-card implementation written using *HCLOOC* of matrix multiplication for Intel Xeon Phis.
- The very first out-of-card implementation of matrix multiplication for Xilinx FPGAs.

1.3 Thesis Structure

The structure of this thesis is as follows. In Chapter 2, we review the evolution of HPC machines, efforts on performance and energy modelling and optimization for modern heterogeneous platforms, and the existing work to solve the bi-objective optimization problem for performance and power/energy consumption on HPC systems. We also describe the existing out-of-card implementations for accelerator kernels. In Chapter 3, we present a novel data partitioning algorithm for performance optimization of data-parallel applications on heterogeneous HPC platforms and evaluate its efficiency and scalability on HPC clusters. In Chapter 4, we present new methods for dynamic energy modelling and optimization on heterogeneous HPC platforms, and then experimentally evaluate their accuracy and efficiency. In Chapter 5, two model-based data partitioning algorithm are proposed, one for solving bi-objective optimization for execution time and dynamic energy, and the other optimizes two objectives execution time and total energy. In Chapter 6, we will describe a programming library enabling out-of-card implementation for accelerator kernels on heterogeneous computing platforms, and then experimentally examine its performance on two Nvidia GPUs, one Intel Xeon Phi and one Xilinx FPGA. Finally, Chapter 7 concludes the thesis.

Chapter 2

Background and Related Work

In this chapter, first, the evolution of HPC machines is reviewed. Next, we overview data partitioning techniques as well as state-of-the-art load-balancing methods. After that, efforts on performance and energy modelling for modern heterogeneous platforms are surveyed. Then, we will study the existing approaches to achieve bi-objective optimization for performance and power/energy consumption on modern HPC systems. Last, research work presenting out-of-card implementations for accelerator kernels will be overviewed.

2.1 Heterogeneous HPC Platforms

Multiple Instruction, Multiple Data (MIMD) parallel computers are referred to a broad type of high-performance computing platforms. These machines consist of a number of independent processing elements which enable the execution of different instructions on different pieces of data. A processing element refers to either a multicore CPU or an accelerator such as a Graphics Processing Unit (GPU), an Intel Xeon Phi (Xeon Phi), a Field Programmable Gate Array (FPGA), and etc. There are two classifications of MIMD machines: a). *shared memory*, e.g. multicores and CPU-accelerator heterogeneous platforms, and b). *distributed memory*, e.g. computational clusters. Shared memory systems take advantage of a common data space (memory) shared between

all processing elements. However, each processing element has its own private memory in distributed memory platforms which interacts with others using message passing techniques. The time taken for access to the memory can be identical for all processing elements, called Uniform Memory Access (UMA), or for some processing elements is longer than others, referred to as NUMA. Shared memory systems can be designed in both types UMA and NUMA, but the memory access time in distributed platforms is NUMA.

Now, we are going to review the evolution of MIMD HPC platforms. For more than three decades prior to mid-2000s, computer users came to expect performance doubling every 18 month due to Moore's law and Dennard scaling [4, 5]. In that period of time, microprocessors, consisting of several single-core CPUs, such as those in the Intel Pentium family and the AMD Opteron family, brought swift performance increase and cost reduction in computer applications for more than two decades [74]. To keep pace with escalating performance requirements for enterprise and parallel applications, processor manufacturers primarily responded by increasing the processor clock speed. However, increasing the clock speed leads to unprecedented energy consumption and heat dissipation issues; and, by 2004, computer designers hit the power wall caused by unprecedented problems stemming from increasing power consumption and increasing power density (amount of power dissipated per unit area, which represents the heat dissipation). The power problem was caused primarily by the breakdown of Dennard scaling, a scaling model whereby the power density of a transistor-based processor of a unit area remains constant due to voltage and current scaling down with the length of the transistor. Up until 2004, moving to a smaller transistor process meant frequency could be increased for no increase in heat dissipation. The breakdown of Dennard scaling meant frequency scaling was no longer economical [6]. The chip fabrication industry turned to multicore CPU architectures to address this problem of increased power consumption and power density. Frequency scaling was abandoned in favour of multiple processors per chip. In fact, excellent overall processing performance was achieved by reducing clock speed while increasing the number of processing units; and the consequent reduction in clock speed can lead to lower heat output and greater efficiency [7].

Apart from multicore processors, we witnessed the emergence of many-core accelerators, such as GPUs and Intel Xeon Phis, which take advantage of a great number of simple smaller cores. General purpose computing on GPUs became practical with the appearance of programmable shaders and floating point support circa 2001. The release of CUDA [75] by Nvidia in 2007 and subsequent availability of high-quality high-performance scientific libraries (CUBLAS, CUFFT, etc.) fuelled the extensive use of GPUs by the scientific communities, especially for matrix computations. In addition to enhanced programmability, their superior energy efficiency (performance per watt or energy per flop) compared to multicore CPUs for certain class of HPC applications has been one of the prominent reasons behind their rapid adoption by the HPC community. Intel entered the accelerator market by launching their Xeon Phi coprocessors based on its Many Integrated Core (MIC) architecture in 2013. Its key selling point compared to programming on GPUs is the reusability of existing parallel approaches (and reduced portability concerns) due to its standard x86 programming model. Therefore, because of their progressively improving programmability and better energy efficiency, accelerators have become an integral part of HPC platforms addressing the twin critical concerns of performance and energy consumption. FPGA is now believed to be a serious contender to these accelerators. However, its smooth adoption by the HPC community has been hindered by its poor programmability.

To summarize, extreme-scale HPC computing systems today feature tight integration of multicore CPU processors and accelerators (GPUs or Intel Xeon Phis) empowering them to provide not just unprecedented computational power but also to address the well established critical concerns of power and energy efficiency. It is apparent that heterogeneity has become a common and inseparable attribute of high-performance computing platforms. The current Top500 list [8], which is a harbinger of extreme-scale platforms, contains 126 systems with Intel/AMD multicore CPUs and Nvidia GPU accelerators (Fermi/Kepler/Pascal), and 31 platforms with Intel Xeon Phi accelerators. There are 4 hybrid systems integrated with Nvidia and Intel Xeon Phi coprocessors. This hybrid and highly heterogeneous nature of the systems is now widely perceived to be a leading forerunner to achieving exascale objectives. It is

envisaged that futuristic extreme-scale platforms will feature nodes that will contain multiple types of accelerators and will utilize only a subset of these that are best suited to execute specific classes of applications and to run different portions of a hybrid application.

Therefore, we believe that this paradigm shift towards hybrid architectures is now becoming firmly entrenched and will be pervasive as we move towards and into the exascale future.

2.2 Data Partitioning on HPC Platforms

Despite significant efforts to design and produce powerful HPC platforms, exploiting the parallelism capability of these machines depends on developing efficient parallel applications. To design and implement a parallel algorithm, we need an *algorithm model*. An algorithm model determines a way of structuring a parallel algorithm by selecting a *decomposition* and *task mapping* technique and applying the appropriate strategy to minimize interactions [1]. Namely, decomposition refers to splitting a problem into smaller sub-problems where each will be assigned to a task to run in parallel. Mapping is a mechanism expressing how tasks are assigned to different logical processing elements for parallel execution. There are various ways to undertake task mapping. But, a good mapping is one that meets the desirable objective(s). It should be mentioned these objectives in this thesis are minimum execution time and minimum energy consumption. There exist two algorithm models commonly used to design parallel algorithms:

- **Data parallel:** In this model, the data is decomposed into some partitions and each partition is given to a task.
- **Task parallel:** It relies on a graph called the task dependency graph which expresses dependencies among tasks and reveals the interaction pattern of tasks in a given parallel application. The graph is a Directed Acyclic Graph (DAG) so that the nodes represent tasks, and the directed edges determine dependencies amongst them.

At the followings, we first explain different decomposition techniques and then study task mapping approaches.

Decomposition is the only way to implement concurrency in HPC applications. The common problem decomposition techniques, as the first step in designing a parallel application, are [1]:

- Recursive decomposition,
- Data decomposition,
- Exploratory decomposition,
- Speculative decomposition.

In this thesis, we concentrate on the powerful and commonly used approach, data decomposition, which suits the data parallel algorithm model. In this decomposition model, the data on which the computations are performed is partitioned, and this data partitioning is then used to induce a partitioning of the computations into tasks. This approach decomposes workload into continuous partitions, which consequently enhances data locality on processing elements. As an example, consider a dense matrix A with a size of $n \times n$ multiplying with a vector b. Figure 2.1 shows the decomposition of these dense matrix-vector multiplication into n partitions. In this example, the computation is divided among n tasks where the workload to calculate the i-th cell of the result vector y involves the i-th row of A and whole b, which is given to one task.

After decomposing computation into some tasks, we need to undertake a mapping solution which assigns the tasks to processing elements so that its parallel execution time and energy consumption are minimized. To achieve this goal, we should alleviate the overheads of running parallel applications on HPC platforms, which come from inter-process interactions and process idle-time. The first metric is out of the scope of this thesis, and we focus on process idle-time. Consider a parallel application where some processes of which finish their computations while others are working on the problem. In this case, some processors should wait idly before starting a new task.

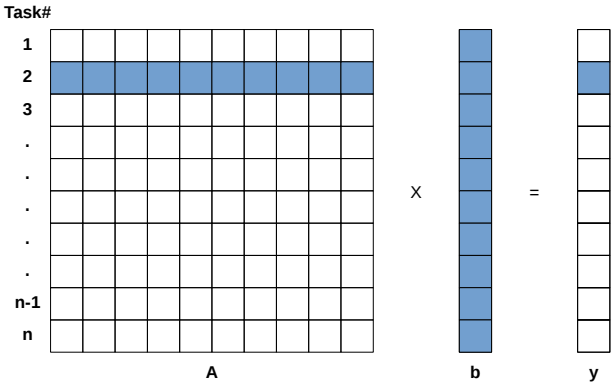

Figure 2.1: Decomposition of a matrix-vector multiplication into n partitions, where n represents the number of rows in the matrix. Each cell of the result vector y is calculated by one task [1].

To address the process idle-time, one solution is to keep the load balanced among processors. A balanced application does not waste processor cycles in waiting at points of synchronization and data exchange. This maximizes the utilization of processors which consequently decreases the execution time and even the energy consumption of parallel applications. We will study the state-of-the-art load-balancing techniques in Section 2.2.1

2.2.1 Load-balancing in HPC platforms

Generally, load-balancing algorithms are classified into several categories:

- Static,
- Dynamic,
- Centralized,
- Non-centralized,
- Task-queue,
- Predicting-the-future.

The algorithm model, which represents the characteristics of tasks and the interaction patterns among them, determines which category is suitable for mapping.

Static algorithms, such as those based on data partitioning [76, 77, 78, 15, 79], use *a priori* information about the platform and parallel application such as the knowledge of task sizes, the size of data associated with tasks, the characteristics of inter-task interactions, and even the algorithm model. Since relying on accurate performance models which predict the future execution of applications, these approaches are also known as *predicting-the-future*. They are especially appropriate for applications where data locality is important because they do not require data redistribution. Nevertheless, these algorithms may not be suitable for non-dedicated platforms, where load changes over time. In addition, these algorithms are not appropriate for parallel applications with non-deterministic task sizes that the execution time required by the tasks varies significantly. Generally, finding an optimal static mapping is a kind of NP-complete problem and cannot be solved in polynomial time. That is why algorithms practically deploy some heuristics to find quite acceptable sub-optimal solutions to the optimal static mapping problem.

Dynamic algorithms, such as work stealing and task scheduling [80, 81, 82, 83, 84], balance load by migration of fine-grained tasks between processors during their execution. These algorithms are suitable for applications with dynamic task generations. They do not need any priori information about execution but may impose large communication overhead due to data migration especially in NUMA and distributed memory platforms. The cost of overhead may also outweigh the advantages of dynamic mapping. Applying static partitioning for the initial step can enhance dynamic algorithms due to its provably near-optimal communication cost, bounded tiny load imbalance, and lesser scheduling overhead [83, 85]. Dynamic algorithms based on graph partitioning approaches are proposed by [86, 87] for adaptive scientific computations where two objectives, data migration costs and interprocessor communication, are considered.

In non-centralised algorithms [88, 89], the load is migrated locally between neighbouring processors, while in centralised ones [90, 91, 92], global load

information is deployed for finding appropriate load distribution. Although non-centralised approaches converge slower, centralised ones typically incur relatively higher overhead. The centralised algorithms can be further subdivided into two groups: *task-queue* [91] and *predicting-the-future* [90, 92].

2.3 Performance and Energy Models of HPC Platforms

Algorithms, for performance maximization and energy consumption minimization, require performance and energy models of parallel applications to solve data partitioning problems and perform task mappings. We will overview the performance and energy models for HPC platforms in this section. The models are classified in two main groups listed below:

1. **Analytical-based models:** A vast majority of these models are based on linear or non-linear regression techniques and use operating system reported data, microarchitectural parameters, code parameters, or Performance Monitoring Counter (PMC) to estimate the performance and energy consumption. PMCs are special purpose registers available in modern computing architectures to store software and hardware activities counts.

2. **Non-analytical-based models:** These models take the real-life behaviour of applications running on HPC platforms to model their performance and energy consumption. This type of modelling relies on the direct measurement of the desired parameter.

2.3.1 Models for Performance

In this section, we survey analytical and non-analytical (real-life) approaches for modelling and maximizing performance on multicore CPUs and accelerators.

Analytical-based models

An empirical non-linear performance model for microprocessors is proposed in [93]. It is highlighted that the cycle-accurate simulation to model performance is very costly because simulators are too slow. That is why empirical analytical-based modelling can be an alternative to the pure simulation-based ones. They build a separate model for each program in SPEC CPU2000 benchmark suite. The model involves 9 architectural parameters which are selected empirically using simulation techniques.

The Fast and Accurate Simulation Environment (FASE) [94] is a trace-driven framework designed to facilitate performance prediction and system design by finding the ideal configuration for a specific set of applications. It is suitable for MPI-based C and Fortran HPC applications running on clusters of CPUs. FASE characterizes the behaviour of an application to obtain the accurate representations of its execution. The collected data are communication information, computation information, memory accesses and disk I/O. The collected data is fed into the simulation environment to predict performance and explore the various system configurations.

Roofline model facilitates an insightful visual performance estimation for multicores [95] and accelerators [96, 97]. This model takes into consideration floating-point performance, operational intensity, and memory performance to estimate an upper bound on feasible performance in terms of piece-wise linear models. This model can be used to enhance performance via adjusting the operational intensity of applications.

Baghsorkhi et al. [98] predict the performance of general-purpose applications running on GPU architectures based on an analytical model. The execution time of GPU kernels is estimated based on non-linear regression and analysing workflow graphs. A workflow graph represents an abstract interpretation of a GPU kernel which is used to estimate the maximum parallelism can be achieved without violating local resource usage in GPUs. The model can be used by auto-tuning compilers or by programmers to find bottlenecks in their codes.

Zhang and Owens [99] present an instruction-level performance model

which estimates the execution time of applications running on Nvidia GeForce 200-series GPUs. It provides detailed quantitative performance information on three main architecture components in GPUs including the instruction pipeline, shared memory access and global memory access. The proposed method is suitable for the bottleneck detection and the performance analysis of GPU programs. Programmers and architects can take advantages of this model to predict the benefits of potential program optimizations and architectural improvements.

A performance modelling framework for heterogeneous platforms, including GPUs and FPGAs, is proposed in [100]. It is based on the patterns of computation and memory accesses that occur within an application. The model estimates the performance of an application and its benefit from a device based on machine and application characteristics. A benchmark suite is used to recognize the machine characteristics, and data footprints of an application express its characteristics. The footprint reveals the application's computing pattern.

Shen et al. [101] proposed an analytical-based workload partitioning approach for heterogeneous platforms, consisting of multicore CPUs and GPUs. The workload partitioning problem is analytically modelled using the equation $\frac{W_G}{W_C} = \frac{P_G}{P_C} \times \frac{1}{1+(O/W_G)\times(P_G)/Q}$, where W_G and W_C are respectively workload sizes of GPU and CPU, P_G and P_C respectively represent the processing throughputs of GPU and CPU, O is data-transfer size, and Q represents data-transfer bandwidth. $W_G = \beta \times W$ and $W_C = (1 - \beta) \times W$, where β represents the fraction of workload assigned to the GPU, and W is total workload size. The workload partitioning is predicted by solving the workload partitioning model to determine β.

In [102], a fine-grained workload partitioning framework, called FinePar, is proposed. They use the linear regression technique to predict the performance of OpenCL applications running on CPUs and GPUs. The performance of CPUs and GPUs are modelled using the linear equation $performance = C_1 \times AW + C_2 \times VW + C_3 \times \log NW + C_4 \times \log SW + C_5$, where the average workload for a work-item (AW), the variance of the distribution of non-zero elements across the rows (VW), the number of work-items in the computation

domain (NW) and the size of the whole workload (SW) are input parameters.

Rosales et al. [103] predict the execution time of applications, running on Intel processors, as a non-linear formulation, $execution\_time = \frac{W_{cpu}}{R_{cpu}} + \frac{W_{bw}}{R_{bw}}$, where W_{cpu} is the amount of work the application requires from CPU, W_{bw} is the amount of work the application requires from memory, R_{cpu} represents the floating point processing rate, and R_{bw} determines the memory bandwidth. The application is executed on a machine with different processor speeds and bandwidths then its W_{cpu} and R_{bw} are analytically predicted for target machines.

PyPassT [104] is an analysis based modelling framework. It models execution time and resource utilization for HPC platforms, including CPUs and accelerators. It statically analyses an application source code, written in C with OpenACC directives, to capture its runtime behaviour. The application is then run on a simulated target HPC architecture to analysis its performance. The simulated machine analytically models computation time and memory cost. Because of static analysis of code, the tool cannot capture runtime decisions, such as the number of iterations of a loop or a branching probability.

Ding et al. [105] introduce a linear performance model to obtain the scaling performance behaviours and the potential performance bottlenecks. The execution time of an parallel MPI application, consisting of n kernels, can be estimated using the equation $execution\_time = \sum_{i=1}^{n}(T\_comp_i + BF\_mem_i \times T\_mem_i) + BF\_comm \times T\_comm + T\_others$, where T_comp is total computation time, BF_mem is non-overlapping part for loading data from local memory, T_mem represents total memory time, BF_comm expresses the ratio of non-overlapped communication time, T_comm is average communication time, and T_others represents the time taken by initialization and finalization.

Non-analytical-based models

Over the past years, load-balancing algorithms, aiming performance optimization on parallel platforms, have tried to take into consideration the real-life behaviour of applications executing on these platforms. This can be perceived by looking at the evolution of performance models for computation used in

these algorithms. The most straightforward model used is Constant Performance Model (CPM). In this model, each processor in an HPC platform is represented by a positive constant number such as normalized cycle time, normalized processor speed, average execution time, task computation time, etc. to characterize the speed of an application [106, 11, 12, 13, 107, 108, 109]. A common feature of these approaches is that they do not assume any dependency between the performance of a processor and the size of the workload, running on it.

Due to the advances in hardware technology and the emergence of modern heterogeneous platforms, the performance of processing elements became so complicated which completely deviates from the conditions assumed by the CMP-based model. In fact, the CMP-based model is too simplistic to accurately enough estimates the performance of modern heterogeneous platforms. As an alternative, the most advanced load-balancing algorithms use Functional Performance Models (FPMs), which are application-specific and represent the speed of a processor by a continuous function of problem size [14, 110, 15]. These FPMs capture precisely the real-life behaviour of applications executing on nodes consisting of uniprocessors (single-core CPUs), and it is supposed that the shape of the function is so smooth that satisfies one of the following assumptions:

1. Along each of the problem size variables, the function is monotonically decreasing,
2. There exists point x such that
   - On the interval $[0, x]$, the function is
     - concave,
     - monotonically increasing, and
     - any straight line coming through the origin of the coordinate system intersects the graph of the function in no more than one point.
   - On the interval $[x, \infty)$, the function is monotonically decreasing

Table 2.1: Specifications of three clusters of the Grenoble site from Grid'5000. All nodes are connected with InfiniBand 20G & 40G.

Cluster	Processor	Cores	Memory	GPU
Adonis 1-10	2.27 Xeon E5520	8	24GB	Tesla T10
Adonis 11-12	2.4GHz Xeon E5620	8	24GB	Tesla C2050
Edel	2.27GHz Xeon E5520	8	24GB	-
Genepi	2.5GHz Xeon E5420 QC	8	8GB	-

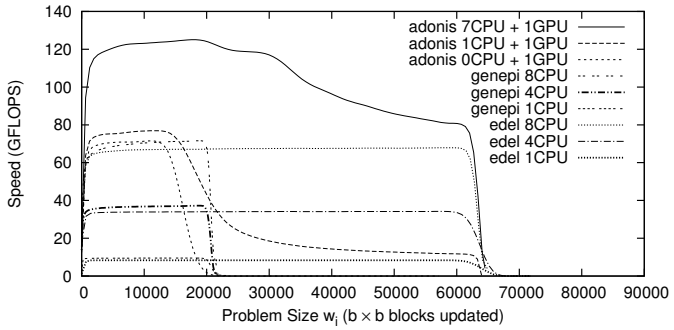

Figure 2.2: Functional performance models (FPM) of BLAS DGEMM on a number of nodes from Grid'5000 Grenoble site [2].

Figure 2.2 represents speed functions of blocked BLAS DGEMM on three different clusters, Adonis, Genepi and Edel, of the Grenoble site from Grid'5000 [2]. In this experiment, the result matrix C is partitioned into $b \times b$ blocks, and problem sizes, w_i, are in number of $b \times b$ blocks. All nodes have 8 CPU cores, and 12 of them also involve Nvidia Tesla GPUs in Adonis cluster. Table 2.1 shows the specifications of these three clusters.

The shapes of performance profiles are smooth with no significant variation. The speed functions drop rapidly where the workload is not fitting into the available main memory of the processors, and therefore paging is required.

Due to the complex nodal architecture of modern HPC systems, with tightly integrated processors, severe resource contention, NUMA, and highly hierarchical design, the performance profiles of parallel applications executing on these platforms involve lots of variations and violate the conditions assumed by the proposed FPM-based algorithms proposed in [21, 22, 23, 24, 25, 26, 27, 28, 29]. In this case, applying the state-of-the-art load-balancing algo-

rithms returns sub-optimal and even non-optimal workload distributions. To deal with this challenge, novel model-based algorithms have been proposed which are able to find optimal workload distribution on state-of-the-art homogeneous systems where the proposed approaches make no assumptions about the shapes of performance profiles.

Lastovetsky et al. [16] propose an optimization algorithm based on an advanced performance model of computation (FPM) by using novel load distribution to minimize the computation time. First, they experimentally build the performance profiles of the application for a wide range of problem sizes separated by a minimum granularity. They then employ this function and its connected visual picture to distribute workloads unevenly between *homogeneous* groups of cores of an Xeon Phi co-processor, consequently load-imbalancing the application, to achieve performance optimization. This is the first effort that the load-imbalancing technique is applied for the workload partitioning of a parallel application and minimizing the computation time of its parallel execution. It should be mentioned that they propose no general partitioning algorithm in this work. Lastovetsky et al. [3, 18] focus on the importance of workload distribution as a decision variable and propose such general model-based methods and algorithms for minimization of not only the time but also the energy of computations for the most general performance and energy profiles of data parallel applications executing on *homogeneous* multicore clusters. Figure 2.3 presents the speed function of FFTW executing on a homogeneous multicore server. The server includes an Intel Haswell E5-2670 v3 @ 2.30 GHz, consisting of 24 identical cores, and 64 GB of RAM. This application invokes 24 parallel threads to perform Fast Fourier Transpose on square matrices of size $m \times m$. The complex nodal architecture has incurred drastic variations in the performance profiles of the application. They formulate the performance and energy optimization problems and present efficient algorithms of complexity $O(m^2 \times p^2)$ solving these problems where m is the cardinality of the discrete sets representing the speed/energy functions and p is the number of available processors. The memory complexity of the algorithms is $O(n \times p^2)$. Unlike load-balancing algorithms, optimal solutions found by these algorithms may not load-balance an application.

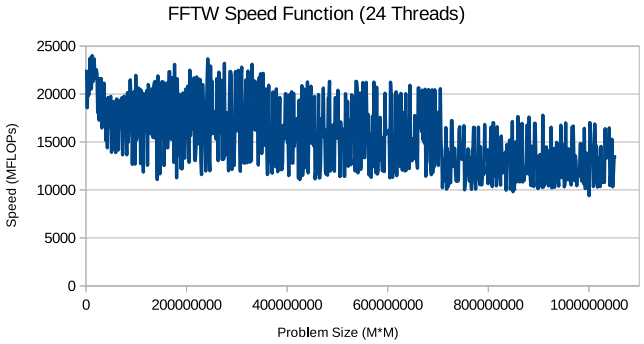

Figure 2.3: Speed function of FFTW running 24 threads to calculate Fast Fourier Transpose of $m \times m$ square matrices on a multicore processor Intel Haswell E5-2670. [3].

Apart from [16, 3, 18], which have explored in-depth the deterministic and reproducible performance variations for bound applications, Zhang et al. [111] also report significant non-deterministic variations for applications that are not bound to the cores of the executing multicore platform.

2.3.2 Models for Energy Consumption

In this section, we overview analytical and non-analytical (real-life) methods for modelling and optimizing power/energy for multicore CPUs and accelerators. These methods generally model the energy consumption of an application in two ways:

- **Explicit:** In the explicit approach, energy consumption is modelled directly with no reference to power.
- **Power-based:** These approaches construct power and performance models of applications. Since $Energy = Execution\_Time \times Power$, the energy consumption of applications can be calculated by using their power and performance models.

Analytical-based models

In this section, we first review efforts have been done to analytically model power/energy consumption on multicore CPUs. Then, the purposed models for HPC platforms including CPUs and accelerators will be surveyed.

SimplePower [112] and Wattch [113] are two fine-grained and cycle-accurate simulators which analytically estimates energy consumption and power dissipation of microprocessors. The components considered in modelling are: ALU and the other modules in a pipelined instruction set architecture, including memory and bus system.

IBM's PowerTimer [114] provides detailed component-level power consumption based on empirical techniques. To estimate the power consumption of an architectural unit, it measures the power consumption of the same unit on a real platform and then scales it according to variations in size and design.

Fan et al. [32] present a model estimating the power consumption of servers. This model expresses a linear relation between CPU utilization and power consumption. It also supports CPU DVFS technique. Another component-level and linear-based approach is proposed in [36] which models the energy consumption of CPUs using five contributors: bus traffic, cache misses, CPU temperatures, environment temperatures and bandwidths for disk read and write.

Lively et al. [37] present power-predictive models for hybrid applications (MPI/OpenMP) based on PMCs. They rank each performance counter using Spearman's rank correlation and then eliminate any counter below a given threshold. They then calculate the Principal Component Analysis (PCA) of the remaining performance counters and select ones with the highest PCA coefficients. After employing this elaborate statistical methodology, 40 PMCs remain for modelling.

Now, we overview power/energy model for HPC platforms. Rofouei et al. [38] estimate energy consumption on CPU-GPU systems using the multiplication of execution time with average power consumption of a device. Therefore, $E_{CPU} = t_{CPU} \times P_{avg-CPU}$, and $E_{GPU} = t_{GPU} \times (P_{avg-GPU} + P_{idle-CPU}) + E_{transfer}$, where t_{CPU} and t_{GPU} respectively represent CPU and GPU usages,

$P_{avg-CPU}$ and $P_{avg-GPU}$ are the average power consumed by CPU and GPU, and $E_{transfer}$ is the amount of energy consumed for data transfer between CPU and GPU.

Multicore Power, Area, and Timing (McPAT) [115] is a framework modelling power, area, and timing for multicore and manycore processors. It considers most of the fundamental components in multicore processors such as cores, interconnects, shared caches and memory controllers. Later, Zhao et al. [35] use the framework to obtain the power breakdown of different components in AMD and Nvidia GPUs. Lim et al. [116] highlight that power consumption of a GPU can be represented as the summation of the power consumptions of all modelled components. They use McPAT to estimate the power consumption of GPU components and adjust their model.

A component-level power consumption model is proposed in [39]. It analytically models the power consumption of 12 components in GPUs, from ALU to memory units. The power consumption is then estimated as the sum of power consumptions of all these components along with their access rates.

Nagasaka et al. [40] aim to estimate the power consumption of CUDA kernels running on GPUs. The model is based on linear regression and utilizes 13 PMCs counting computational and memory utilization.

A power consumption model for GPUs is presented in [117]. It is based on linear regression tree and random forest methods. The random forest method is used to select the parameters that are the dominant contributors to the power consumption. The model collects 22 runtime characteristics including 18 types of operations and 4 architecture parameters.

A power consumption model is presented in [41] which use the trickle-down effect of events in the processors. In this technique, a subset of performance-related events within a microprocessor is used to model power consumption outside of the microprocessor. The model uses local performance-related events within a microprocessor to iteratively model the power consumptions of six sub-systems: microprocessor, chipset, memory, I/O, disk and GPU.

AMG (Activity-based Model for GPUs) [42] is a power consumption model for GPUs. The power is model using the equation $Total\ power\_consumption = Idle\ Power + Runtime\ Power$. $Idle\ Power$

represents the power consumed by a GPU when it is turned on but no kernel is running, and $Runtime\ Power$ is calculated as $Runtime\ Power = \sum_{i=1}^{e}(N_{SM} \times P_{u,i} \times U_{u,i}) + B_{u,i} \times U_{u,i}$. N_{SM} is the number of components, such as floating point, shared and global memories, $P_{u,i}$ represents the power consumption of an active component, e is the number of architectural component types, $B_{u,i}$ is the base power of a component and $U_{u,i}$ represents utilization.

GPUSimPow [118] is an analytical-empirical power simulation framework for general purpose computation on GPUs (GPGPUs). It is capable to precisely simulate the power consumption during the execution of GPGPU workloads, as well as, estimate multiple characteristics of a hypothetical GPU architecture such as chip area, gate leakage and peak dynamic power. Therefore, this simulation enables hardware architects to evaluate design choices early from a power perspective, and GPU programmers to optimize power consumption from a software perspective. It models power as $P = \alpha C V_{dd} \Delta V f_{clk} + V_{dd} I_{Short-circuit} + V_{dd} I_{leakage}$, where the activity factor, α, represents the percentage of the circuit's capacitance being charged during switching. The first term is the dynamic power that is spent charging and discharging capacitive loads when the circuit switches state. The second term of the equation is the short-circuit power being consumed when both pull-up and pull-down networks in a CMOS circuit are on for a short amount of time.

Song et al. [119] present a power model for kernels running on Nvidia Fermi C2075 GPUs. They utilize 10 PMCs selected by applying the Pearson correlation method. They model energy consumption as a product of average power times execution time.

Kestor et al. [120] present a System Monitor Interface (SMI) between the OS and the user runtime that accounts for each core's power consumption. It is based on a regression analysis of core activity. They select significant PMCs, contributing to consumed power, using squared correlation coefficients. The final model considers the number of integer instructions, stalled cycles, last level cache misses and the number of floating point operations.

Choi et al. [43] proposed an arch-line model which is an energy-based analogue of the time-based roofline model presented in [95]. The model visualizes energy and power consumptions based on algorithm-related param-

eters, including arithmetic and memory operations and computation intensity of algorithm; as well as the machine characteristics, such as the time and energy costs per operation or per word of communication. They experimentally evaluated the model on real CPU and GPU platforms.

Shao and Brooks [44] demonstrate an instruction-level energy model for Intel Xeon Phi processors. It characterizes the energy consumption of each instruction (EPI) by using PMCs. To estimate EPIs, memory behaviour, the number of active cores and the number of active threads per core are taken into account. Then, runtime performance counter statistics compute the breakdown of instruction. In the end, the total energy of the workload is calculated by multiply the runtime instruction counts with the corresponding EPI.

Jarus et al. [121] present system-wide energy consumption models for servers, which is based on the analysis of performance counters. They use variants of linear regression to remove PMCs that do not improve the average model prediction error. They classify HPC applications with similar characteristics and then build application-specific models for them. They use decision trees for finding an appropriate model for a given application.

Al-Khatib et al. [122] presents an operand-value-based model to estimate the dynamic energy consumption of FPGAs. They characterize the amount of energy consumed for the execution of an instruction, with zero operands, which is called the base energy of the instruction. Then, the impact of operands on the instruction energy variance is determined. The energy consumption of an instruction is calculated using $E_i = E_{base}(i,j) + k(operand\ value\ property).EV(i)$, where $E_{base}(i,j)$ is the base energy of a given instruction i following instruction j, $EV(i)$ represents the maximum energy variance of instruction i, and k is a factor determining the fraction of the maximum energy variance of instruction i given the operand values.

Shahid et al. [55] propose a novel criterion called *additivity* to determine a subset of PMCs that can potentially be considered for reliable energy predictive modelling for an Intel Haswell-E5-2670 CPU. This criterion is based on the experimental observation that the energy consumption of a serial execution of two applications is the sum of energy consumptions observed for the individual execution of each application. They believe that a linear predictive

energy model is consistent if and only if its predictor variables are additive in the sense that the vector of predictor variables for a serial execution of two applications is the sum of vectors for the individual execution of each application. They show that lots of PMCs, used in energy predictive models, are not additive, and therefore, bring into question the reliability and reported prediction accuracy of these models.

Non-analytical-based models

In this section, we review works evaluating proposed methods for modelling power/energy and efforts which use the direct measurement of power/energy.

Kamil et al. [123] study power measurements for a number of computational loads on the small and large scale HPC platforms. Power efficiency (the ratio of performance per watt) is considered as a metric for comparison. It is also highlighted that measuring power in HPC platforms is not straightforward because more than one system shares the same metered circuit. They list several different methods for measuring power usage which differ in the tools used for measuring power consumption and in the places where valid measurements can be collected. These methods are: (i) *inline meters*, (ii) *clamp meters*, (iii) *integrated meters*, and (iv) *power panels in power distribution units (PDUs)*. They have concluded the power consumption of a large-scale system can be accurately obtained from power measurements of smaller subsets of the system. In addition, they mentioned that it is essential to measure power consumption during running a suitable workload and predictor models based on CPU power cannot be accurate.

Rivoire et al. [124] compare five power modelling techniques based on resource utilization over a variety of machines and benchmarks. Four out of five models are based on OS-reported CPU and disk utilization. However, the fifth model considers CPU PMCs as well as the OS-reported parameters. According to the empirical results, they highlight that the PMC-based approach will be increasingly necessary for accurate power prediction. But, since PMCs used in the model parameter set may not be the same across different processor families (Intel, AMD), they put the generality and portability of the PMC-based

model under question.

McCullough et al. [53] demonstrate linear-based power modelling approaches shows high prediction error in modern computing platforms because of inherent complexities such as multiple cores, hidden device states and large dynamic power components. They show power prediction errors can reach as high as 150 percent and propose direct measurement as an alternative to analytical-based techniques to deal with the inherent complexities arise by modern architectures.

O'Brien et al. [52] study proposed models for power and energy prediction on the highly heterogeneous and hierarchical node architectures in modern HPC platforms. They come up with the idea that the inherent complexities, such as resource contention on Last Level Cache (LLC), NUMA and dynamic power management, make analytical-based approaches less-accurate enough to model performance and energy on modern HPC systems. They highlight the shortcomings of models to accurately and comprehensively estimate the power and energy consumptions by taking into account the hierarchical and heterogeneous nature of these tightly-integrated high performance computing systems. Finally, they conclude that direct measurement is the only accurate way to model the energy consumption of HPC platforms.

Lastovetsky and Reddy [3, 18] propose a model-based energy optimization algorithm on tightly integrated multicore CPUs. Due to inherent complexities (contentions on shared resources and NUMA), they highlighted that the shapes of energy profiles get so complicated that cannot be modelled using linear techniques. They studied the real-life profiles of single and multi-thread applications and concluded as the number of threads increases, the fluctuations in the energy profiles also increase. Figure 2.4 shows the dynamic energy profile of FFTW application on a homogeneous multicore server including an Intel Haswell E5-2670 v3 @ 2.30 GHz, involving 24 identical cores, and a main memory size of 64 GB. This application runs 24 threads to compute parallel Fast Fourier Transpose of $m \times m$ square matrices. Due to the inherent complexities, they used direct energy measurement, rather than analytical modelling techniques, to build real-life energy profiles of parallel applications. In this model, the dynamic energy consumption is represented by a non-linear

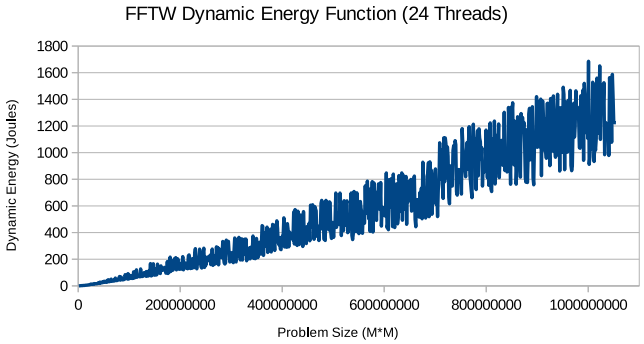

Figure 2.4: Dynamic energy profile against problem size for FFTW application running 24 threads to compute Fast Fourier Transpose of $m \times m$ matrices on a multicore processor Intel Haswell E5-2670. [3].

and non-convex function of problem size.

2.4 Performance and Energy Bi-objective Optimization on HPC Platforms

In this section, we will highlight a number of state-of-the-art approaches to achieve the bi-objective optimization for performance and energy consumption on modern HPC platforms. The proposed methods are generally classified into two broad categories: *system-level* and *application-level*.

2.4.1 System-level Methods

System-level methods target the optimization of several objectives of the system or the environment on parallel platforms such as clouds computing infrastructures, data centres, etc. In this section, we will focus on works taking into consideration the two leading objectives performance and energy consumption. A substantial feature of these approaches is to deploy application-oblivious and heuristic techniques to model these two objectives of applications. We will mention the parameters and decision variables which are used in each model and the relationships between the objectives and decision vari-

ables.

A parallel method to solve a bi-objective optimization problem for performance and energy consumption in cloud computing infrastructures is presented in [61]. It deploys a genetic algorithm to find bi-objective solutions. The parameters, input to the algorithm, are the task computation cost (w) and the communication costs between two tasks. The supply voltage (V) of the processor is the only decision variable. The consumed energy of computations is modelled as a polynomial function of $V^2 \times w$.

Mohammadi Fard et al. [62] propose a multi-objective case study for performance, energy consumption, reliability and economic cost for scientific workflows executing on heterogeneous computing environments. It takes two parameters which are computation speeds of processors and the bandwidths of communication links connecting a pair of processors. Mapping of tasks is considered as the decision variable. The energy consumption of computations is estimated as the cube-root of clock frequency.

In [63], a heuristic-based approach is presented which address energy efficiency and Quality of Service (QoS) optimization for resource management in data centres. The decision variable set involves clock frequencies and the number of VMs. The energy consumption is modelled as a linear function of CPU utilization.

Kessaci et al. [10] propose a three-objective algorithm that reduces the energy consumption, CO2 emissions and increases the generated profit of a cloud computing infrastructure. The algorithm uses a genetic algorithm to find optimal solutions. Input parameters comprise the number of processors used in the execution of an application, the execution time of an application and the deadline for completion of the application. The decision variable is the arrival rate. The energy consumption is estimated as a product of execution time and average power consumption, which is calculated as $\alpha \times f^3 + \beta$, where f is the clock frequency.

A performance and energy optimization algorithm is presented in [64] for applications executing on heterogeneous HPC systems. The impact of nine different parameters on performance and energy consumption of machines are considered. These parameters include the number of cores, the number

of threads, DVFS levels, cache size and thermal design power (TDP). Decision variable in this work is workflow scheduling.

Kolodziej et al. [47] propose a genetic algorithm considering twin objectives of performance and energy consumption for applications executing in green grid clusters and clouds. The performance is modelled as a function of processor speed. DVFS level comprises the only decision variable of this algorithm. Energy consumption is estimated using the equation $\gamma \times V^2 \times f \times t_e$, where γ is a constant for a processor, V is the supply voltage, f is the clock frequency, and t_e is the estimated completion time.

2.4.2 Application-level Methods

Application-level methods essentially aim to optimize applications for performance and energy consumption. In addition, this type of methods relies on application-level models for predicting the performance and energy consumption of applications. In this section, we exclusively focus on three features of each method: i). Type of optimization, ii). Parameters and decision variables and iii). The relationship of two objectives performance and energy consumption with the parameters and decision variables. In addition, this category of methods can be further classified, based on the scope of optimization which is targeted by application-level bi-objective algorithms, into: i). intra-node optimization and ii). both intra-node and inter-node optimizations.

Intra-node Methods

An intra-node optimization method for clusters of DVFS-capable AMD nodes is presented in [65]. There are three input parameters including: the ratio of the application slowdown to the CPU slowdown, memory pressure, which depends on memory operations retired and L2 cache misses, and slack, which predicts communication bottlenecks.

Ahmad et al. [48] aim to minimize the makespan (job completion time) and the energy consumption of computational-intensive scientific problems through task scheduling onto homogeneous and heterogeneous multicore pro-

cessors. Input parameters involve: computational cycles, DVFS levels, and the architecture of core.

Choi et al. [125] prepose an bi-objective optimization algorithm based on roofline models presented in [95] and [43] for performance and energy consumption. They extend the roofline model for energy by considering three more factors: memory hierarchy access costs, power caps and the measurement of random memory access patterns.

Aba et al. [67] present an approximation algorithm to minimize both makespan and the total energy consumption in parallel applications running on a heterogeneous resources system. They use three parameters in this work: the computation cost (w) of a task, the execution frequency of a processor (f) and the communication cost between each pair of processing elements. The decision variable is task scheduling. The makespan of a given task is calculated as a function of $\frac{w}{f}$, and its consumed power is estimated using the equation $w \times f^2$. They ignore all solutions that their energy consumptions exceed a given constraint and then find the solution with minimum execution time.

Inter-node and Intra-node Methods

Subramaniam et al. [49] deploy multi-variable regression techniques to make a trade-off between performance and energy consumption of the high-performance LINPACK (HPL) benchmark. In this work, several models are presented where the final model contains four parameters: the problem size, N, the block size, NB, the number of process rows and processors columns in the process grid, P, Q. This approach gives a single solution in case the problem size and number of processors are fixed. However, our algorithm, will be proposed later in Chapter 5, gives a set of globally Pareto-optimal solutions. Decision variables considered in this work are: the number of nodes, the number of threads and DVFS levels.

Song et al. [68] aims to quantify energy consumption improvements in data-intensive parallel applications running on homogeneous platforms using an iso-energy-efficiency model. The energy improvement of parallel over se-

quential application is studied using three decision variables: clock frequency, level of parallelism, and problem size.

An optimization approach is presented in [69] studying energy savings at the algorithm level. Using classical and Strassen matrix multiplication and the direct n-body problem, they prove there is a region of perfect strong scaling in energy. Thus, for a given problem size n, the energy consumption remains constant as the number of processors p increases and the runtime decreases in proportion to p. The performance is modelled as a linear function of the costs of computations and communications. The energy consumption is estimated using a linear function of costs for computations, communications, and static power.

Inadomi et al. [50] experimentally study and analyse chip manufacturing power variations and show how these variations lead to power and performance inhomogeneity. They propose a *variation-aware* power budgeting algorithm that improves performance under a power constraint. Inputs variables are: a power constraint and a Power Variation Table (PVT), which is application independent and constructed once per system. The table contains the manufacturing variability on a given platform. The power variations of each target application are modelled as a linear function of PVTs. The decision variable is the CPU frequency which maximizes the application performance under a given power constraint. It should be mentioned that our proposed variation-aware approach deploys real profiles, to accurately model the performance and energy consumption of applications and builds globally Pareto-front solutions.

Gholkar et al. [51] present another variation-aware solution for performance-power optimization on limited power budget platforms. The algorithm takes as inputs: the maximum number of processors on a machine, N_{max}, the power budget of the machine, $P_{m\ c}$, and the number of processors requested by a job, n_{req}. The decision variable is the CPU frequency and the optimal number of processors for a job. They suppose that the power consumption of the interconnect is zero, and DRAM power consumption is ignored. However, we will propose a bi-objective variation-aware optimization approach for performance and energy which uses real profiles to model the

performance and energy consumption of parallel applications running on a heterogeneous cluster and takes into account the energy consumptions of all components in the platform.

The impact of memory hierarchies on the performance and energy consumption of parallel applications is studied in [70]. Performance and energy are modelled as two linear functions of data size. These functions are used for maximizing performance and minimizing energy in parallel processing of divisible loads. In this study, the number of processors is taken into account, however, the other parameters are fixed.

Tarplee et al. [71] consider optimizing two conflicting objectives, the makespan and total energy consumption of all nodes in an HPC platform. They employ linear programming (LP) and divisible load theory (DLT) to compute tight lower bounds on the makespan and energy of all tasks on a given platform. Using this formulation, they then generate a set of Pareto-front solutions. The decision variable is task mapping. The parameters used to formulate the problem involve: the number of task types, the number of machines, ETC, which is a matrix representing **E**stimated **T**ime to **C**ompute each task on each machine, and APC, another matrix involving the **A**verage **P**ower **C**onsumption of each task on each machine where generally obtained from historical data in real environments.

Gabaldon et al. [72] introduce a multi-objective genetic algorithm to compromise between the makespan and energy consumption of parallel applications running on Federated Cluster environments. The decision variable is task scheduling or mapping. The makespan is defined as the elapsed time between the submission of the first job until the finalization of the last one. The energy consumption is modelled using the equation, $C \times CT + I \times IT$ where C is the energy consumed by a node when it is computing, I when it is idle, CT determines the computing time of the node and IT represents idle time.

Chakrabarti et al. [73] propose a data partitioning scheme addressing the execution time and dirty (non-renewable) energy consumption optimization on heterogeneous clusters. They use progressive sampling and then deploy function fitting techniques to estimate execution time given the input data size.

They estimate renewable energy availability using the PVWATTS simulator. The obtained results from the simulator are combined with the predicted execution time to obtain the objective function of dirty energy consumption. The decision variable is partition size distribution, however, the proposed approach does not take into account the real-life behaviour of applications and also use a linear programming formulation to solve the bi-objective optimization problem. In fact, the approach returns the optimal solution when the execution time is approximately linearly related to the problem size and the variations in the renewable energy availability are minimal so that the availability is close to the mean energy supply. Henceforth, it is not a variation-aware solution.

Manumachu et al. [45, 46] experimentally study the performance and energy profiles of real-life data-parallel applications on state-of-the-art multicore CPUs and demonstrate that there exists a complex (non-linear and even non-convex) relationship between these two objectives and problem size. They propose algorithms to solve performance-energy optimization problem for applications executing on homogeneous multicore platforms. The algorithms employ only one decision variable, the workload distribution. They takes as inputs discrete functions of performance and dynamic energy consumption against problem size and returns the globally Pareto-optimal set of solutions. These approaches are applicable to solve the problem for homogeneous platforms including identical processors. However, our algorithm, will be proposed in Chapter 5, is able to solve the bi-objective optimization problem on modern heterogeneous HPC platforms without any assumption on the shape of profiles (permanence and energy) and the number and type of processors.

2.5 Summary

In the previous sections, we reviewed efforts for performance and energy modelling as well as the proposed bi-objective optimization methods for these two metrics.

In the single-core processors era, analytical approaches were able to precisely estimate the performance and energy consumption of applications using

a few architectural and program parameters. However, the tight integration of multicore CPUs with many-core accelerators incurs new complexities, such as contentions on shared resources and NUMA. These complexities make the proposed analytical models less-accurate. In the following list, we enumerate some of the reasons that make analytical-based approaches inappropriate for modelling:

1. Unprecedented complexities, including resource contention and NUMA, have made the performance and energy consumption profiles of parallel applications running on modern HPC platforms too complicated. That is why these analytical models have been reported to be inaccurate [53, 52, 54, 55].

2. Apart from a few variation-aware algorithms for performance and energy optimization on homogeneous HPC platforms [3, 45, 18, 46], all proposed methods assume a linear relationship between workload size and performance and between workload size and energy consumption. Nevertheless, regarding some aforementioned efforts [24, 50, 51, 52, 3, 16, 111, 45, 18, 46], one can conclude that profiles on modern HPC platforms are highly non-linear that makes the relationship between workload size and performance and between workload size and energy consumption so complex, non-linear and even non-convex. Therefore, application-oblivious and eventually workload size-oblivious models cannot reveal the exact real-life behaviour of applications on modern heterogeneous platforms. Using this type of algorithms, which ignore variations in performance and energy profiles, consequently leads to sub-optimal solutions for performance and energy optimization problems.

3. Some of these approaches rely on too many parameters which makes them too sophisticated to understand and implement [93, 102, 105, 37, 39, 40, 117, 42, 118, 64].

4. Most analytical models use PMCs and OS-reported data (software PMCs) to obtain input parameters. In addition, there are analytical approaches which depend on the analysis of applications' source codes

written in specific programming languages [94, 99, 102, 104, 40]. This reliance on PMCs and specific programming language puts the portability of the analytical models under question.

5. A few analytical models use simulators to estimate performance and energy. These models can be criticised for the following reasons:

   (a) Simulators are not able to keep pace with the fast-changing hardware architectures,

   (b) They abstract the real hardware which results in losing accuracy,

   (c) They are not portable, and cannot be used to model the performance and energy consumption of any kind of application on any desirable platform,

   (d) Their modelling cost is very expensive due to the low speed of cycle-accurate simulations especially when the workload size is very large.

According to the aforementioned reasons, analytical-based methods for performance and energy modelling on modern HPC platforms cannot be accurate enough whereby solving optimization problems using these methods may result in sub-optimal solutions. Therefore, we would like to summarize that direct measurement is the only way to accurately model the performance and energy consumption of modern HPC platforms as functions of problem size. These functions consider both manufacturing and application variations.

We then aim to use these real-life functions as inputs to our proposed model-based and variation-aware algorithms to solve performance, energy and bi-objective optimization problems for data-parallel applications running on clusters of *heterogeneous* platforms. These algorithms involve only one decision variable, which is the workload distribution.

To the best of our knowledge, there is no model-based data-partitioning approach for performance and energy optimization on modern *heterogeneous* platforms which takes *real-life behaviour* and *variations* of applications into account. Although a few research efforts [50, 51, 111] have taken into account variations, they model performance and energy analytically, rather than

real-life measurements, and ignore workload distribution as a decision variable. Some research [14, 110, 15, 21, 22, 23, 24, 25, 26, 27, 28, 29] has considered both heterogeneity and real-life functions, but these approaches are not variation-aware. As a result, applying them on modern HPC platforms will lead to sub-optimal solutions. In spite of considering both variations and real-life behaviour of applications, some previous works [3, 16, 45, 18, 46] are only applicable to solve optimization problems on homogeneous platforms.

2.6 Out-of-card Computation on Accelerators

In this section, we first look at notable works that implement out-of-card kernels for accelerators and then review open source and vendor developed out-of-card libraries released for accelerators.

2.6.1 Out-of-card Implementation of Accelerator Kernels

Gu et al. [126] present an out-of-card implementation of FFT kernel for a single GPU. The authors co-optimize both CPU-GPU data transfer via PCI-E bus and on-GPU computation for 1D, 2D and 3D FFTs by using the Cooley-Tukey decomposition framework. The framework is used for decomposing a large-sized FFT into smaller sub-FFTs, which are then transferred to the GPU in batches. A recursive kernel is proposed to compute on-card FFT. To achieve high throughput on the CPU-GPU data channel, a blocked buffer technique for 1D FFTs is developed. The effect of sub-array size on data transfer performance is also studied in this paper. They find that PCI-E bus bandwidth decreases when sub-array size decreases due to the consequent increase in the number of *cudaMemcpyAsync* calls. To deal with small sub-arrays and to increase the PCI-E bus bandwidth, continuous sub-arrays are buffered and then transferred by using a single *cudaMemcpyAsync* call.

Mu et al. [127] introduce an out-of-card algorithm for LU decomposition. The proposed approach is based on the left-looking factorization on GPU/CPU platform where it uses both the host memory and the hard disk for out-of-card computations.

In 2012, Zhong et al. [128, 129] propose an out-of-card implementation for matrix multiplication routine (DGEMM) for Nvidia GPU. However, the implementation places some constraints on the dimensions of the matrices that are allowed in the matrix multiplication. In Chapter 6, we will remove these constraints and apply additional optimizations to improve the performance of our proposed out-of-card library.

An out-of-card dense matrix multiplication implementation for CPU-GPU platforms similar to [128, 129] is presented in [130]. They perform matrix decomposition according to peak bandwidth of PCI-E links and the bandwidth required by an application.

Sabne et al. [131] present a computation splitting technique that automatically adjusts the number of pipeline stages to improve the performance of out-of-card implementations on multiple GPUs attached to the same host CPU.

Shirahata et al. [132] present out-of-card techniques for large-scale graph processing applications for heterogeneous GPU-based clusters.

Out-of-core implementations for large dense singular value decompositions (SVD) for CPU architectures are proposed in [133, 134] that use disk storage in cases the problems are too large to fit into the main memory.

Yamazaki et al. [135] propose out-of-card algorithms to factorize a symmetric indefinite matrix for CPU and GPU architectures.

2.6.2 Out-of-card Libraries for accelerator kernels

In this section, we overview the released libraries supporting out-of-card computations. The CUBLAS-XT library [136] provides a set of Basic Linear Algebra Subprograms (BLAS) routines that utilize multiple GPUs connected to the same motherboard. It uses CUDA streams [137] and events to efficiently manage data transfers across PCI-Express bus and kernel invocations on the GPUs. The routines in the library also support out-of-card operation where the size of the matrices is limited only by the system memory size. However, we show in this work that our out-of-card implementation of the DGEMM routine out-performs that provided in CUBLAS-XT library.

SciGPU-GEMM [138] is a library of wrapper functions to help use the GEMM routines from CUBLAS on GPUs with limited memory and no double precision hardware.

HPL-CUDA [139] is a library for high-performance computing Linpack benchmark for CUDA. It does not contain out-of-card implementation for level-3 BLAS matrix multiplication routine.

MAGMA (Matrix Algebra on GPU and Multicore Architectures) [140, 141] is a library providing out-of-card algorithms for dense LU, Cholesky and QR factorizations for CPU-GPU platforms.

Chapter 3

A Novel Data-Partitioning Algorithm for Performance Optimization of Data-Parallel Applications on Heterogeneous HPC Platforms

Modern HPC platforms have become highly heterogeneous owing to the tight integration of multicore CPUs and accelerators (such as GPUs, Xeon Phis, or FPGAs) which empower them to maximize performance, as a dominant objective. Due to this inherent characteristic, processing elements contend for shared on-chip resources, such as Last Level Cache (LLC), interconnect, etc., and shared nodal resources, such as DRAM, PCI-E links, etc. This severe resource contention and also Non-Uniform Memory Access (NUMA) have posed serious challenges to model and algorithm developers.

As an example, Figure 3.1 presents the speed functions of 2D FFT application on a hybrid node, including an Intel Haswell multicore CPU, one Nvidia K40c GPU and one Intel Xeon Phi 3120P. Each accelerator is connected to a dedicated host core via a separate PCI-E link. One can observe significant fluctuations in the performance profile of the application.

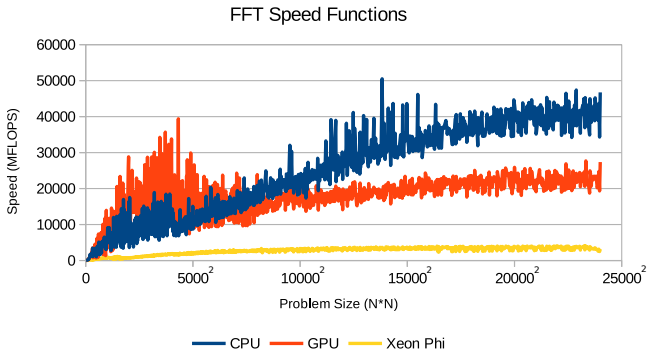

Figure 3.1: Speed functions of heterogeneous 2D FFT application executing on a heterogeneous node including an Intel Haswell multicore CPU, one Nvidia K40c GPU and one Intel Xeon Phi 3120P.

As explained in Chapter 1, the most advanced load-balancing algorithms use functional performance models (FPMs), which are application-specific and represent the speed of a processor by a continuous function of problem size with an almost smooth and convex shape [14, 110, 15]. However, performance profiles of modern HPC systems involve lots of variations and violate the conditions assumed by the proposed FPM-based algorithms proposed in [21, 22, 23, 24, 25, 26, 27, 28, 29]. Therefore, load-balancing data partitioning algorithms based on FPMs may not return optimal solutions.

To summarize, the complexities (resource contention, NUMA, accelerator-specific limitations, etc.) have introduced new challenges to optimization of data-parallel applications on these platforms for performance. Due to these complexities, the performance profiles of data-parallel applications executing on these platforms are not smooth and deviate significantly from the shapes that allowed state-of-the-art load-balancing algorithms to find optimal solutions.

In this chapter, we explain how to model the computational performance of hybrid platforms, using abstract processors. Then, the problem for optimization of data-parallel applications on modern heterogeneous HPC platforms for performance is formulated. After that, we propose a novel model-based data partitioning algorithm, which minimizes the execution time of computations in the parallel execution of an application via load imbalancing. This algorithm

takes as input the problem size n, a set of p discrete speed functions corresponding to p available heterogeneous processors and considers only one decision variable, which is workload distribution. The algorithm requires individual performance profiles of all the processors and does not make any assumption about the shapes of these functions. We prove the correctness of the algorithm and its complexity to be $O(m^3 \times p^3)$, where m is the cardinality of the input discrete speed functions.

We experimentally demonstrate the optimality and efficiency of our algorithm using two data-parallel applications, matrix multiplication and fast Fourier transform, on a heterogeneous cluster of nodes where each node contains an Intel multicore Haswell CPU, an Nvidia K40c GPU and an Intel Xeon Phi co-processor.

3.1 Modelling Computational Performance of Hybrid Platforms

In this section, we explain how to model the performance of heterogeneous platforms in terms of a set of speed functions [15]. These functions are built empirically and are application and platform specific. Therefore, models must be built for each application on each unique processing element.

The performance of an application running on a processor is modelled using a discrete profile, which is named speed function. Each function contains a set of data points where each point represents the computational speed of the application ($s(w)$) for a given problem size w. The computational speed of a given application on a processor is defined as the number of operations executed by the processor divided by the application execution time, which is represented by $t(w)$. Since the computational complexity of an application is a measure to estimate the useful work involved in processing the application, we can obtain the number of operations using the computational complexity of the application, which is represented by $C(w)$. Therefore, the computational speed of a given application is calculated as:

$$s(w) = \frac{C(w)}{t(w)} \tag{3.1}$$

Consider the matrix multiplication of two large dense square matrices as an example. Suppose each matrix consists of $n \times n$ float elements. The computational complexity of a straightforward algorithm to calculate the result matrix is $C(n) = O(n^3)$. Therefore, the useful work (the number of operations) to compute the result matrix involves around $2 \times n^3$ floating point operations (FLOP).

Apart from the number of operations, we need the application execution time, $t(w)$, to build each data point in the performance profiles. There are fine-grained high precision timers in any computer which can be used to measure the execution time of each application running on processors.

In the single-core era, all hardware resources are exclusively utilized by one application. However, a heterogeneous data-parallel application, which consists of a number of kernels (generally speaking, multithreaded), runs in parallel on different parts of a hybrid platform. In general, due to tight integration and severe resource contention in heterogeneous platforms, the load of one computational kernel in a given hybrid application may significantly impact the performance of others to the extent, preventing from the ability to model the speed of each kernel in hybrid applications individually. Henceforth, computational kernels cannot be considered independent and their performance (execution times) should not be measured separately.

To address this issue, in this work we restrict our study to such configurations of hybrid applications, where individual kernels are coupled loosely enough to allow us to build their individual speed functions with the accuracy sufficient for successful application of the optimization algorithms, proposed later in this thesis. To achieve this, we only consider configurations where no more than one CPU or accelerator kernel is running on the corresponding device. Then, each group of cores executing an individual kernel of the application is modelled as an abstract processor [142] so that the executing platform is represented as a set of heterogeneous abstract processors. Each abstract processor solely constitutes the processing elements and resources

which are involved in the execution of a given application kernel on it. We make sure that the sharing of system resources is maximized within groups of computational cores representing the abstract processors and minimized between the groups. This way, the contention and mutual dependence between abstract processors are minimized.

Since the abstract processors contain CPU cores that share some resources such as main memory and QPI, they cannot be considered completely independent. Therefore, the performance of these loosely-coupled abstract processors must be measured simultaneously, thereby taking into account the influence of resource contention. That is, the data points for a problem size in the speed functions of an application are experimentally built so that the same workload is simultaneously executed on all the abstract processors in the platform.

Take a multi-accelerator NUMA node, which is named *HCLServer01*, as an example. It contains an Intel Haswell multicore CPU consisting of 24 physical cores with 64 GB main memory, whose specification is shown in Table 3.1. In addition to the multicore CPU, the node integrates two accelerators, Nvidia K40c GPU and Intel Xeon Phi 3120P, whose specifications are shown in Tables 3.2 and 3.3 respectively. Each accelerator is connected to a dedicated host core via a separate PCI-E link.

Figure 3.2 represents the block diagram of *HCLServer01* and its abstract processors. The first abstract processor contains 22 (out of total 24) CPU cores executing the multi-threaded CPU kernel. These cores are highlighted in dark blue in the figure. The second abstract processor comprises the Nvidia K40c GPU, its dedicated host CPU core executing the GPU kernel along with the PCI-E link connecting the host to the accelerator. The abstract processor is highlighted in orange. And finally, the third abstract processor, which is highlighted in red, consists of Intel Xeon Phi 3120P coprocessor, its dedicated host CPU core executing the Xeon Phi kernel along with its PCI-E link. The dedicated host CPU core is responsible for sending data from host to accelerator, kernel invocations on the accelerator and then copying results back from the accelerator to host. Therefore, the pair consisting of an accelerator and its dedicated host core executing one accelerator kernel is modelled by an ab-

Table 3.1: Specification of the Intel Haswell multicore CPU.

Technical Specifications	**Intel Haswell E5-2670V3**
Thread(s) per core	2
No. of cores per socket	12
Socket(s)	2
NUMA node(s)	2
CPU MHz	1200.402
L1d cache	32 KB
L1i cache	32 KB
L2 cache	256 KB
L3 cache	30720 KB
NUMA node0 CPU(s)	0-11,24-35
NUMA node1 CPU(s)	12-23,36-47
Processor base frequency	2.30 GHz
Total main memory	64 GB DDR4
Memory bandwidth	68 GB/sec
TDP	240 W
Idle Power	61 W

Table 3.2: Specification of the Nvidia K40c GPU.

Technical Specifications	**Nvidia K40c**
No. of processor cores	2880
Base clock	745 MHz
Boost clock(s)	810 MHz, 875 MHz
Total board memory	12 GB GDDR5
L2 cache size	1536 KB
Memory bandwidth	288 GB/sec
Memory I/O	384-bit GDDR5
Memory clock	3.0 GHz
TDP	235 W
Idle Power	16 W
Idle Power (Persistence mode)	68 W

stract processor. The kernel executing on an accelerator uses all its cores. The execution time of a kernel in the GPU and Xeon Phi abstract processors includes the times of data transfer between the accelerators and their host cores. Since there should be a one-to-one mapping between the abstract pro-

Table 3.3: Specification of the Intel Xeon Phi 3120P.

Technical Specifications	**Intel Xeon Phi 3120P**
No. of processor cores	57
Base frequency	1.10 GHz
Total main memory	6 GB GDDR5
L2 cache size	28.5 MB
Memory bandwidth	240 GB/sec
Memory clock	3.0 GHz
TDP	300 W
Idle Power	91 W

cessors and computational kernels, any hybrid application executing on the server in parallel should consist of three kernels, one kernel per computational device.

As explained earlier, due to existing some shared resources between the abstract processors, they are not independent, and the performance of these abstract processors must be measured simultaneously. We explain how to do this in our experimental methodology presented in Appendix A. It should be noted that while speed functions are built where the data points for the same problem size are obtained simultaneously, during the actual execution of the data-parallel application using the workload distribution determined by the proposed data partitioning algorithm, the problem sizes executed by the abstract processors can be different. This is because different processors can be allocated different problem sizes by our heterogeneous data partitioning algorithm. However, since abstract processors are as loosely coupled as possible, the speeds of execution of these problem sizes simultaneously would not differ significantly from those present in the speed functions; the marginal differences do not imply significantly different execution times. We confirm this to be the case through exhaustive experimentation; synopsis of this is presented in Appendix B.

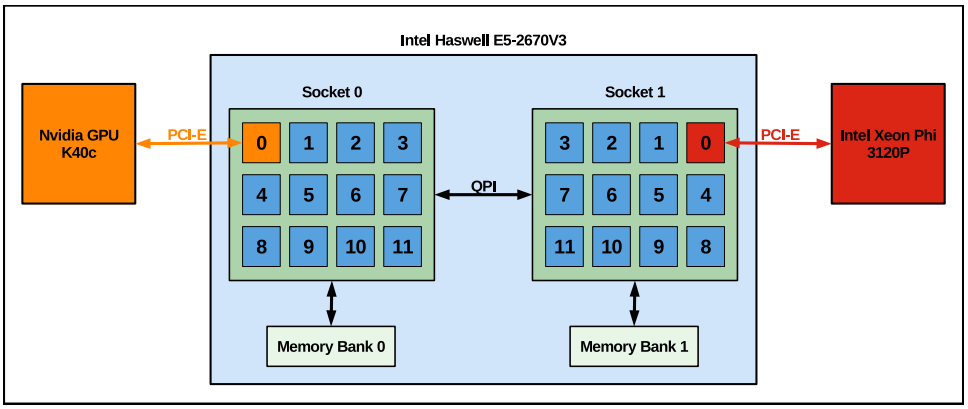

Figure 3.2: Block diagram of HCLServer01 *including an Intel Haswell multi-core CPU, one Nvidia K40c GPU and one Intel Xeon Phi 3120P.*

3.2 Formulation of Performance Optimization Problem

Consider a problem size n executed using p heterogeneous processors, whose speed functions are represented by $S = \{s_0(x), ..., s_{p-1}(x)\}$ where $s_i(x)$, $i \in \{0, 1, \cdots, p-1\}$, is a discrete speed function of cardinality m of processor P_i. The speed $s_i(x)$ for a problem size x for processor i is calculated as $\frac{C(x)}{t_i(x)}$, where $C(x)$ represents the computational complexity for executing the problem size, and $t_i(x)$ is the time of execution of the problem size. Without loss of generality, we assume $x \in \{1, 2, \cdots, m\}$. The performance optimization problem can be then formulated as follows:

Performance Optimization Problem, *HPOPT*(n, p, m, S, X_{opt}, t_{opt}): The problem is to find a partitioning, $X_{opt} = \{x_0, ..., x_{p-1}\}$, of the problem size n using p available heterogeneous processors so as to minimize the computation time of parallel execution of the workload. The parameters (n, p, m, S) are the inputs to the problem. The outputs are X_{opt}, which is the workload distribution, and t_{opt}, which is the optimal execution time. The optimal solution does not necessarily balance the load between processors. We will explain this issue in Section 3.4 This problem can be formulated as an Integer Non-Linear Programming (INLP) problem as follows:

$$
\begin{aligned}
t_{opt} &= \min_{X} \max_{i=0}^{p-1} \frac{C(x_i)}{s_i(x_i)} \\
\text{Subject to} \quad & x_0 + x_1 + ... + x_{p-1} = n \\
& 0 \leq x_i \leq m, \qquad i = 0, \cdots, p-1 \\
\text{where} \quad & p, m, n \in \mathbb{Z}_{>0} \quad \text{and} \quad x_i \in \mathbb{Z}_{\geq 0} \quad \text{and} \\
& s_i(x) \in \mathbb{R}_{>0}
\end{aligned}
\tag{3.2}
$$

It should be noted that the execution time $t_i(x)$ for a problem size x for processor i is calculated as $t_i(x) = \frac{C(x)}{s_i(x)}$, where $C(x)$ determines the computational complexity of execution of the problem size, and $s_i(x)$ is the speed of execution of the problem size.

The objective function in the formulated optimization problem is a function of workload distribution X, $X = \{x_0, ..., x_{p-1}\}$, of a given problem size n between the p processors. For each given X, it returns the time of its parallel execution, which is calculated as the time taken by the longest running processor to execute its workload. Any distribution that minimizes this function is considered optimal as its execution time of the problem size n by the p processors cannot be improved. The number of active processor (processors with a non-zero workload) may be less than p in the optimal workload distribution X_{opt}.

3.3 *HPOPTA*: Algorithm Solving HPOPT

In this section, we present our algorithm, *HPOPTA* (**H**eterogeneous **P**erformance **OPT**imization **A**lgorithm), that solves HPOPT using the branch-and-bound solution method. The bounding criteria in this algorithm are *time threshold* and *size threshold*, which will be explained later.

First, we informally describe the algorithm using an example. The input to the algorithm is a set of discrete time functions, which are derived from discrete speed functions. In the example, consider four heterogeneous processors ($p = 4$), which are available for execution of a problem of size $n = 16$.

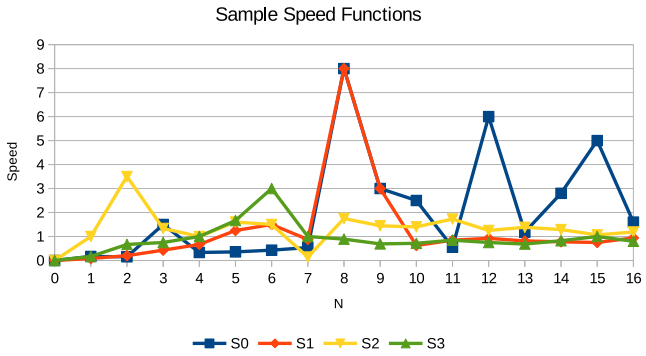

Figure 3.3: Speed functions of a sample application executing on an assumed parallel machine which consists of 4 processors.

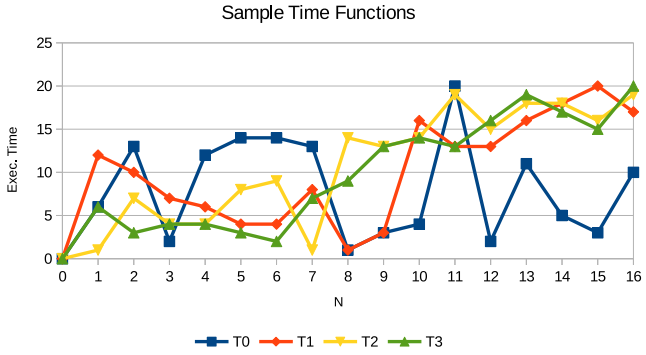

Figure 3.4: The equivalent time functions for the sample speed functions in Figure 3.3.

Figures 3.3 and 3.4 respectively show the sample speed functions, $S = \{s_0(x), \cdots, s_3(x)\}$, and the equivalent time functions, $T = \{t_0(x), \cdots, t_3(x)\}$, of the processors ($m = 16$ in our example). For the sake of simplesity, it is supposed that $s_i(x) = x$, where $i \in \{0, 1, 2, 3\}$. It should be noted that these time functions are samples, which are representative of real-life data-parallel applications.

Figure 3.5 shows the discrete time functions, stored as arrays in non-decreasing order of execution time.

To find the optimal workload distribution, a straightforward approach would be to examine all combinations and select a workload distribution with the minimum computation time of parallel execution of the workload. Figure 3.6

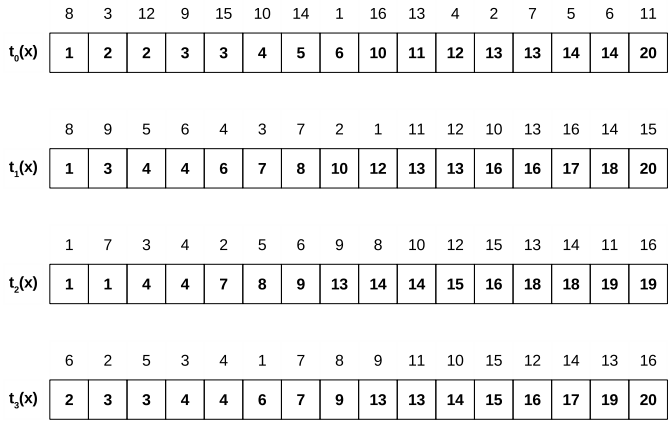

Figure 3.5: Example: The sample time functions, shown in Figure 3.4, which are stored in array data structures. Each array is sorted in non-decreasing of execution time.

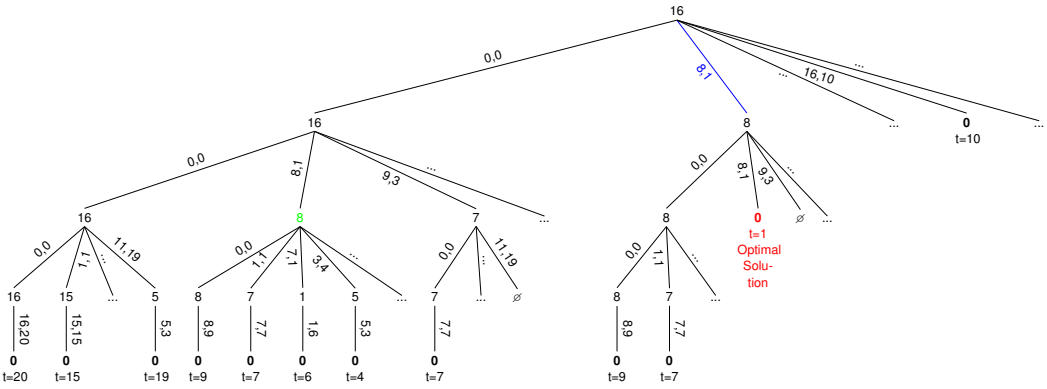

Figure 3.6: Applying naive approach to examine all combinations and select a workload distribution with the minimum computation time of parallel execution of the workload.

shows the tree, which is constructed by such a naive algorithm and contains all the combinations. Due to the lack of space, we only show the tree partially.

The solution tree is constructed from the root, which is the only node at level L_0 of the tree. The value 16, which labels the root node, represents the whole workload to be distributed between 4 processors $\{P_0, P_1, P_2, P_3\}$. Then, 17 problem sizes, including a zero problem size along with all problem sizes existing in the time function $(t_0(x))$, are assigned to the processor P_0 one by one. Although the problem sizes can be given to the pro-

cessor in any order, we assign them in a non-decreasing order of their execution time by the processor. As shown in Figure 3.6, problem sizes $\{0, 8, 3, 12, 9, 15, 10, 14, 1, 16, 13, 4, 2, 7, 5, 6, 11\}$, which have been sorted in non-decreasing order of execution time, are assigned to P_0 one-by-one at level L_0. Therefore, the root node is expanded into 17 children. The value, which labels an internal node at level L_1 (the root's child), represents the remaining workload to be distributed between processors $\{P_1, P_2, P_3\}$.

In its turn, each internal node at level L_1 becomes a root of a sub-tree, which is a solution tree for distribution of the remaining workload between three processors $\{P_1, P_2, P_3\}$. Each edge connecting the root and its child is labelled by the workload assigned to P_0 and its execution time. For example, the blue edge in Figure 3.6 is labelled by $(8, 1)$, which indicates that a problem of size 8 is given to P_0 and it takes one time unit to execute this workload by P_0. The child node connected by this edge is labelled by 8, which is the remaining workload ($= 16 - 8$) to be distributed between processors $\{P_1, P_2, P_3\}$.

In Figure 3.6, the leaf node at level L_1 labelled by 0 represents a solution leaf. In general, any leaf node labelled by 0 represents one of the possible solutions, and the execution time of the corresponding solution is calculated as the maximum of the execution times labelling the edges in the path connecting the root and the solution leaf. For example, the execution time of the solution represented by the leaf labelled by red 0, which is connected to the root by two edges $\{(8, 1), (8, 1)\}$, will be equal to $max\{1, 1\} = 1$. The execution time of the solution represented by the solution leaf at level L_1 will be equal to 10 as it is connected to the root by just one edge $(16, 10)$.

The leaf node at level L_2 labelled by $\varnothing$ is a *no-solution* leaf. The path connecting this node to the root consists of two edges $\{(8, 1), (9, 1)\}$. The corresponding workload distribution results in no-solution because the sum of the problem sizes assigned to P_0 and P_1 will be equal to 17 ($= 8 + 9$), which would exceed the total workload of 16.

In this example, each internal node in the solution tree has either 17 children (or $m + 1$ in general case) or just one child. The child is always a leaf. There are two types of leaves: *solution* leaves, labelled by 0, and *no-solution* leaves, labelled by $\varnothing$. Each internal node at level L_i, labelled by a positive

number w, becomes a root of a solution tree for distribution of the workload w between processors $\{P_i, \cdots, P_{p-1}\}$ and is therefore constructed recursively.

Finally, a distribution minimizing the parallel execution time will be returned as the optimal solution. In this example, the workload distribution $(8, 8, 0, 0)$, represented by the red *solution* leaf and resulting in the execution time of 1, will be returned as optimal.

It is apparent that the complexity of the presented straightforward algorithm is exponential.

We propose an efficient recursive sequential algorithm, *HPOPTA*, of polynomial complexity. *HPOPTA* deploys a number of optimizations to avoid examining all the possible solutions and therefore does not explore all the paths in the tree.

The first step is to sort the discrete time functions, stored as arrays, in non-decreasing order of execution time as shown in Figure 3.5. We then determine the load-equal distribution and its parallel execution time, which is stored in variable τ called the *time threshold*. The load-equal distribution is the distribution where each processor is allocated the same workload of $\frac{n}{p}$ (assuming n is divisible by p). *HPOPTA* will not examine solutions with execution times greater than or equal to the time threshold. In the example, τ will be initialized by 12 ($\max_{i=0}^{3} t_i(\frac{16}{4}) = \max\{12, 6, 4, 4\} = 12$). Therefore, only data points with execution times less than 12 will be considered and form the reduced search space. These data points are shown in gray cells in Figure 3.7. During the execution of *HPOPTA*, the time threshold τ will be updated every time a faster solution is found representing thus the execution time of the currently fastest solution.

HPOPTA then starts examining the solutions in the tree in the left-to-right depth-first order as shown in Figure 3.8. First, processors P_0 and P_1 are allocated zero problem size each, making the workload to be distributed between processors P_2 and P_3 equal to 16. However, this workload exceeds the maximum workload, 15, that can be distributed between these two processors and executed in parallel in less than $\tau = 12$ time units. This maximum workload is associated with level L_2 of the solution tree and called the *size threshold* of this level, σ_2. In general, size threshold σ_i depends on the time threshold,

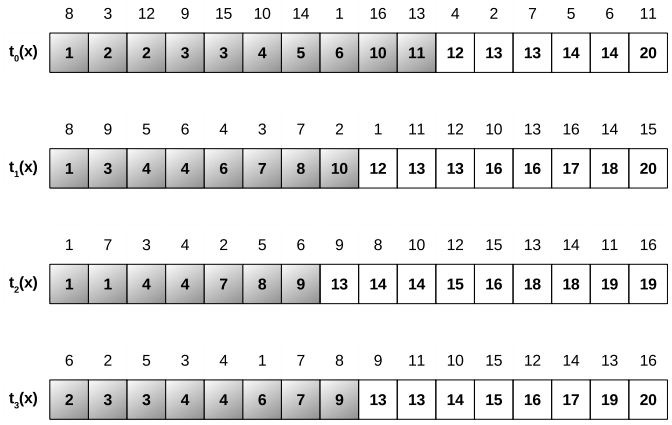

Figure 3.7: Example: Applying load-equal time threshold and removing some data points from the search space.

τ, and is defined as the maximum workload that can be executed in parallel by processors $P_i, \cdots, P_{p-1}$ faster than in τ time units. The vector of size thresholds $\sigma = (\sigma_0, \sigma_1, \sigma_2, \sigma_3)$ can be determined using the time arrays and the current time threshold as follows. The maximum workloads, the execution time of which are less than $\tau = 12$ in the time arrays for processors P_0, P_1, P_2, and P_3, will be 16, 9, 7 and 8 respectively. Therefore, the size threshold of the last level (L_3) will be $\sigma_3 = 8$. The size threshold of level L_2 will be 15 ($= 7 + \sigma_3 = 7 + 8$). Similarly, the size thresholds σ_1 and σ_0 for levels L_1 and L_0 will be 24 ($= 9 + \sigma_2 = 9 + 15$) and 40 ($= 16 + \sigma_1 = 16 + 24$) respectively. Thus, in Figure 3.8, the node labelled by 16 in L_2 cannot lead to solutions, which would be faster than the currently best (load-equal) solution with parallel execution time $\tau = 12$, and therefore this node will not be expanded. So, the red subtree in Figure 3.8 is cut and not explored. We call this key optimization operation *Cut*.

In general, as the algorithm progresses the vector of size thresholds, σ, changes every time the time threshold, τ, decreases. To illustrate how σ changes, we show its value before and after each discussed step of the algorithm. As the *Cut* operation does not change τ, it also will not change σ, as illustrated in Figure 3.8,

Following the left-to-right depth-first order, next node to examine will be

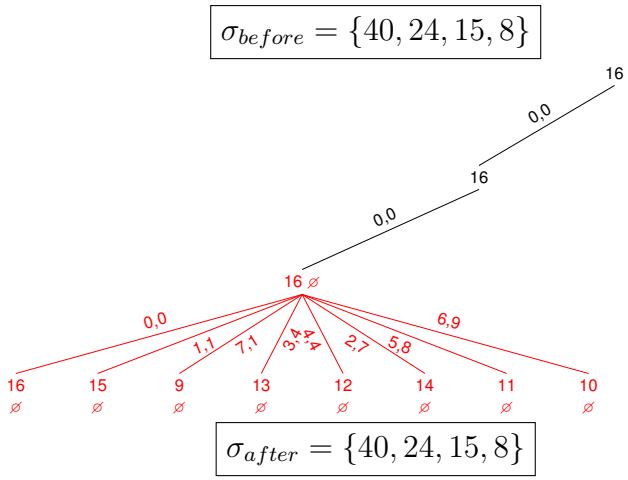

Figure 3.8: Example: Applying size threshold which results in cutting some subtrees, which do not give any solution, from the search tree.

node 8 at level L_2 as shown in Figure 3.9. Proceeding from this node, the algorithm will generate and process solutions (leaves in the tree labelled by 0) in the left-to-right order. For each generated solution, the following operations will be performed:

- The time threshold τ is updated.
- If τ decreases, the data points in the time functions, whose time is greater than or equal to the updated time threshold, are removed from the search space, and the vector σ of size thresholds is updated.
- The solution is saved.
- Backtracking to an ancestor node of the solution is performed. We will explain in detail later how this ancestor node is chosen.

As an example, consider the solution with distribution $\{(0, 0), (8, 1), (3, 4), (5, 3)\}$ and execution time 4 (see Figure 3.9). The time threshold, τ, is updated to 4. Based on the new time threshold, the number of data points to be examined in the time functions is reduced. This is illustrated in the Figure 3.10, where one can see that fewer data points need to be examined compared to Figure 3.7. The vector of size thresholds, σ, is

updated to $\{37, 22, 13, 6\}$. The solution is saved, which includes memorization of the information pertaining to all the levels except for the first and the last. Thus, the information that is memorized is level-specific. For L_1, the saved information includes the problem size assigned to P_1, which is 8, the index of the current element in the corresponding time function, which we call the last examined index and which is equal to 0, and the parallel execution time of the solution for processors $\{P_1, P_2, P_3\}$, which is 4. Saving the last examined index helps *HPOPTA* to resume the exploration of further points from where it was interrupted by backtracking, which will explain later. The same is done for L_2. The saved information includes the problem size assigned to P_2, which is 3, the index of current element in the corresponding time function, which is equal to 2, and the parallel execution time of the solution for processors $\{P_2, P_3\}$, which is equal to 4. We call this key operation, *Save*.

As explained earlier, backtracking to an ancestor node is one of the operations performed after each generated solution. Now, we describe how the ancestor node is chosen for the backtracking. From the leaf pertaining to the current solution, we traverse up the tree to the node with the maximum execution time. The parent of this node will be the backtracking target. For example, again consider the solution with distribution $\{(0, 0), (8, 1), (3, 4), (5, 3)\}$ and execution time 4 in Figure 3.9. For this solution, the node with the maximum execution time will be at level L_2 (the node labelled by 8). Therefore, the algorithm will backtrack to its parent, node 16 at level L_1, as indicated by a blue arc in Figure 3.9. Performing this backtracking effectively means that the algorithm will not generate and process the remaining solution leaves descending from node 8 at level L_2 which are highlighted in red in Figure 3.9.

The reason for this is that the children of the node 8 are examined in a non-decreasing order of time taken by processor P_2 to execute its workload in the corresponding solutions. Therefore, no edge coming out of node 8 after the edge $(3, 4)$ can have a label with the execution time less than 4. This makes further expansion of node 8 meaningless as no solution resulting from this expansion will have execution time less than 4, which is necessary to improve the currently best solution. Therefore, we backtrack to its ancestor, node 16 at level L_1. We will call this key operation *Backtrack*. After backtracking to node

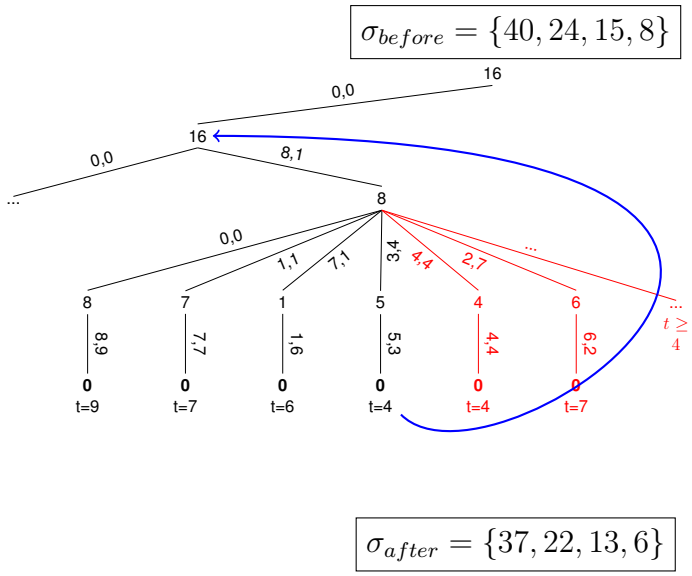

Figure 3.9: Example: Backtracking to the ancestor of the node with maximum execution time, and cutting branches which do not result in any solution better than the solution have found so far.

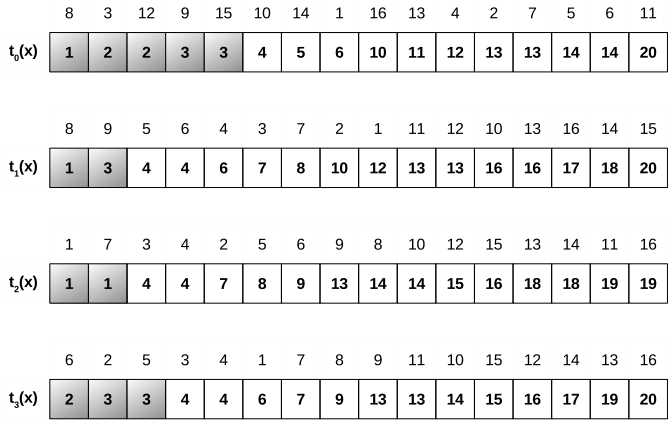

Figure 3.10: Example: Applying the updated time threshold and removing more data points from the search space.

16, the solution saved for the workload 8 at level L_2 becomes *final* because the corresponding distribution of workload of size 8 between processors P_2 and P_3 is the optimal one.

After backtracking to node 16 at level L_1, next node to examine will be node 7 at level L_2. The expansion of this node results in two children as shown in

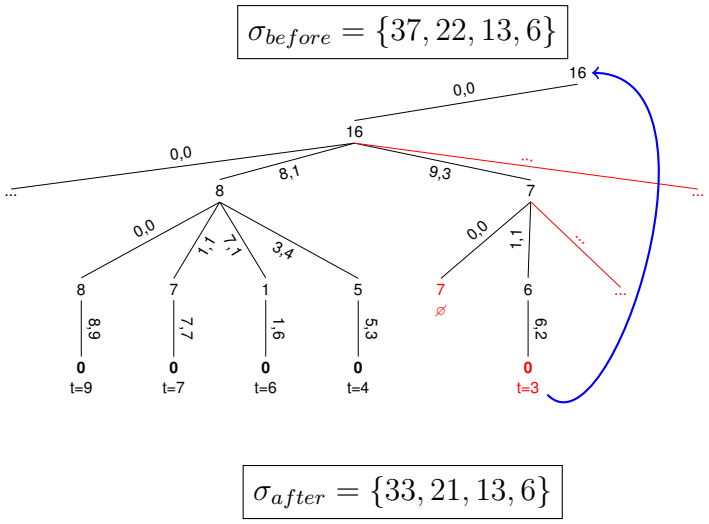

Figure 3.11: Example: Keeping on applying HPOPTA *on the search space.*

Figure 3.11. Giving zero workload to P_2 results in the workload of size 7 at level L_3, which exceeds the size threshold $\sigma_3 = 6$ and therefore results in no-solution. The second child yields a solution, which has the parallel execution time of 3. The algorithm updates the time threshold, τ, making it 3. As the time threshold decreased, the vector of size thresholds is updated to $\{33, 21, 13, 6\}$. The solution then is saved. For L_1, the memorized information includes the problem size assigned to P_1, which is 9, the last examined index which is equal to 1, and the parallel execution time of the solution for processors $\{P_1, P_2, P_3\}$, which is 3. For L_2, the saved information includes the problem size assigned to P_2, which is 1, the index of the current element in the corresponding time function, which is equal to 0, and the parallel execution time of the solution for processors $\{P_2, P_3\}$, which is equal to 2. After this, *HPOPTA* backtracks to the root.

HPOPTA proceeds in this manner from the root until it comes to node 8 at level L_1 as illustrated in Figure 3.12. Here, as the optimal distribution of the workload 8 between processors P_2 and P_3 has been already found and saved, the best solution coming out of node 8 at level L_2 will be just retrieved from the memory. We call this key operation, *ReadMemory*. Since the parallel execution time of the retrieved solution is equal to 4, which is greater than the current time threshold $\tau = 2$, this solution is ignored. The algorithm then

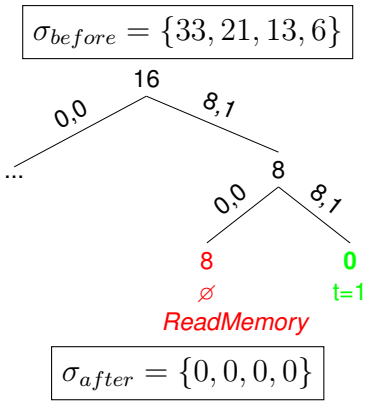

Figure 3.12: Example: Finding the optimal solution and using Mem to find solutions.

moves to the next child, which results in the solution $\{(8,1),(8,1),(0,0),(0,0)\}$ with the parallel execution time of 1. For this solution, the time threshold, τ, is updated to 1. The corresponding reduction of the search space results in the situation when no more data points in the time functions are left for further examination. Therefore, the algorithm terminates.

The optimal execution time is given by the last value of the time threshold. The optimal workload distribution is given by the workload distribution associated with this time threshold. So, there are four key operations in the algorithm, which are a). *Cut*, b). *Save*, c). *Backtrack*, and d). *ReadMemory*.

In Section 3.5, we give a pseudocode of our algorithm, which uses these key operations as the fundamental building blocks.

3.4 *HPOPTA* as a Load Imbalancing Algorithm

In this section, we explain what we mean by load-balancing and load-imbalancing algorithms. *HPOPTA* is a load-imbalancing algorithm. Nevertheless, if the optimal workload distribution load balances the application, then *HPOPTA* finds it.

We define a load-balancing algorithm as one that determines the workload distribution where the problem sizes allocated to the processors are proportional to their speeds. The intuition behind load balancing is that balancing the

application improves its performance in the following manner: a balanced application does not waste processor cycles on waiting at points of synchronization and data exchange, maximizing this way the utilization of the processors and minimizing the computation time.

To find load balance distributions, a straightforward approach is to examine all combinations and select a workload distribution with the minimum difference between the execution times of processors. However, the complexity of this naive approach would be exponential. Lastovetsky et al. [16] studied and presented a formal study of load-balancing algorithms to address this problem. In this work, they show that in order to guarantee that the balanced configuration of the application will execute the workload n faster than any unbalanced configuration, the speed functions $s_i(x)$, characterizing the performance profiles of the processors, should satisfy the condition:

$$\forall \Delta x > 0 : \frac{s_i(x)}{x} \geq \frac{s_i(x + \Delta x)}{x + \Delta x}$$

As explained earlier, the speed $s_i(x)$ is calculated as $\frac{C(x)}{t_i(x)}$. This condition means that the increase of the workload, x, will never result in the decrease of the execution time.

However, in this dissertation, we show that this condition is violated by the performance profiles of the data-parallel applications executing on modern HPC platforms.

HPOPTA is designed to deal with the shapes of performance profiles where the condition is no longer satisfied. We call such an algorithm, *load-imbalancing algorithm*, where it determines the optimal workload distribution that minimizes the execution time of computations of a data-parallel application but which does not load balance the application.

We illustrate using a trivial example. Consider a platform consisting of four abstract processors ($p = 4$) with speed functions presented in Section 3.3. Let the workload to be solved be equal to 31 ($n = 31$). In this example, load-balanced solution is $\{(2, 13), (11, 13), (9, 13), (9, 13)\}$ and the load-balanced execution time therefore is 13. However, the optimal solution found by *HPOPTA* is $\{(9, 3), (9, 3), (7, 1), (6, 2)\}$ with the optimal execution time of

3. It is obvious that the optimal solution does not balance the load between processors.

3.4.1 Problem Dimensions in *HPOPTA*

It should be mentioned that *HPOPTA* is a 1D data-partitioning algorithm. However, *HPOPTA* can be directly applied to 2D or 3D problems where the dimensionality can be reduced to 1D. Consider two examples. Our first example is the execution of MPDATA on Intel multicore CPUs and Intel Xeon Phis [16]. The input data structure to MPDATA is a dense 3D object with dimensions (m,n,l) and size $m \times n \times l$. The dimension l is fixed in real-life simulations. From the experiments, it was observed that the speed of MPDATA varies very little with n for constant m. Therefore, *HPOPTA* can be applied directly to performance optimization of MPDATA where the parameter m is partitioned between the processors. In our second example, we consider the application of *HPOPTA* for optimization of *2D FFT* for performance. A sequential *2D FFT* is computed using row-column or the separable method based on 1D FFTs. Briefly, the row-column method consists of 1D FFTs on rows followed by transpose matrix and then 1D FFTs on rows followed by restoration using transpose matrix. The *1D FFT* computation is optimized using direct application of *HPOPTA*. In our future work, we will develop extensions to *HPOPTA* for optimization of 2D and 3D applications for performance.

3.5 Formal Description of *HPOPTA*

In this section, we describe the pseudocode of *HPOPTA*, which is shown in Algorithm 1. The inputs to *HPOPTA* are: the problem size, n, the number of heterogeneous processors, p, and a array of p time functions, $T = \{T_0, T_1, \cdots, T_{p-1}\}$. T_i represents the time function of processor P_i and consists of m pairs (x_{ij}, t_{ij}), $j \in [0, m)$, where x_{ij} is the j-th problem size in the time function and t_{ij} is its execution time by processor P_i. The outputs are the optimal workload distribution, X_{opt}, and the optimal parallel execution

time, t_{opt}. It should be noted that the number of processors selected by the algorithm in the optimal workload distribution may be less than p.

The algorithm first sorts each time function in non-decreasing order of time (Line 2). It then determines the load-equal distribution. The array, X_{opt}, and the time threshold, τ, are initialized to the load-equal distribution and its corresponding execution time respectively (Lines 3-5). Then the vector of size thresholds, σ, is determined using the function *SizeThresholdCalc* (Line 6).

In line 7, the memorization data structure, matrix Mem, consisting of $(p-2) \times (n+1)$ elements, is initialized. It will save the found solutions for processors $\{P_1, \cdots, P_{p-2}\}$. Then, *HPOPTA* invokes the recursive routine, *HPOPTA_Kernel*, to find the optimal workload distribution.

Function GETTIME(T, x) (called in Line 5) returns the execution time of problem size x in time function T. It returns 0 if x equals 0. It should be mentioned that pseudocodes of all functions used in Algorithms 1 and 2 and the structure of Mem are explained in Appendix B.

Algorithm 1 Algorithm Finding Optimal Workload Distribution of Size n for Maximizing Performance

1: **function** HPOPTA$(n, p, T, X_{opt}, t_{opt})$
INPUT:
Problem size, $n \in \mathbb{Z}_{>0}$
Number of processors, $p \in \mathbb{Z}_{>0}$
Time functions, $T = \{T_0, ..., T_{p-1}\}$,
$T_i = \{(x_{ij}, t_{ij}) \mid i \in [0, p), j \in [0, m), x_{ij} \in \mathbb{Z}_{>0}, t_{ij} \in \mathbb{R}_{>0}\}$.
OUTPUT:
Optimal workload distribution, $X_{opt} = \{x_{opt}[0], ..., x_{opt}[p-1]\}$,
$x_{opt}[i] \in \{\bigcup_{j=0}^{m-1} x_{ij} \cup \{0\}\}, i \in [0, p)$.
Parallel execution time, $t_{opt} \in \mathbb{R}_{>0}$

2: $T \leftarrow T \cup Sort_{\uparrow}(T)$
3: $x_{opt}[i] \leftarrow \frac{n}{p}, \forall i \in [0, p-1]$
4: $x_{opt}[i] \leftarrow x_{opt}[i] + 1, \forall i \in [0, n\%p)$
5: $\tau \leftarrow \max_{i=0}^{p-1}$ GETTIME$(T_i, x_{opt}[i])$
6: $\sigma \leftarrow$ SIZETHRESHOLDCALC(p, T, τ)
7: $Mem[i][j] \leftarrow \emptyset, \forall i \in [1, \cdots, p-2], j \in [0, \cdots, n]$
8: HPOPTA_KERNEL$(n, p, 0, T, \tau, \sigma, NULL, X_{cur}, Mem, X_{opt})$
9: $t_{opt} \leftarrow \tau$
10: **return** (X_{opt}, t_{opt})
11: **end function**

3.5.1 Recursive Algorithm $HPOPTA\_Kernel$

The recursive function, *HPOPTA_Kernel* (Algorithm 2), invokes the core operations, *Cut*, *Save*, *ReadMemory* and *Backtrack*. The level of the tree that is processed in this function is given by c. So, the first invocation of *HPOPTA_Kernel* deals with L_0, the next recursive invocation deals with L_1 and so on. It is important to note that X_{opt} holds the best distribution found so far. The array $X_{cur} = \{x_{cur}[0], x_{cur}[1], \cdots, x_{cur}[p-1]\}$ is used to store problem sizes currently assigned to processors $P_i (i \in [0, p-1])$.

Function *Cut* (given in Appendix B) compares the workload n with the corresponding size threshold σ_c to decide whether to expand the node or cut the subtree at level c (Lines 2-4).

Lines 5-11 process the solutions found in the last level L_{p-1}. When a solution, X_{cur}, is found, the routine *ProcessSolution()* is invoked to perform the following operations :

- If X_{cur} is faster than the current best solution, X_{opt}, the time threshold τ will be reduced to the time of X_{cur} and X_{opt} will be updated by X_{cur}.
- When τ decreases, the vector of size thresholds, σ, is correspondingly updated.
- X_{cur} is memorized by invoking the operation *Save*.
- Using X_{cur}, the index of the level, bk, with the maximal execution time is found. If there are more than one level with this time, the level, which is closer to the root, is chosen.

Line 12 sets idx to -1. Variable idx, ranging from -1 to $m-1$, is used to store indexes of data points in the sorted time functions. If idx is equal to -1, the problem size $x_{i\ idx}$ is set to the zero problem size (Lines 28-30), else $x_{i\ idx}$ is the idx-th problem size in the time function T_i.

Before expanding a node at a given level c to generate distributions of the workload of size n associated with this node, the function *ReadMemory* is called to check if any solution distributing workload n between processors $\{P_c, \cdots, P_{p-1}\}$, is currently saved in Mem and retrieve it if this is the case

(Lines 13-27). The function also updates idx (Line 14) to determine the point from where the examination of data points should be resumed.

A memory cell in Mem saves either optimal or intermediate solution. The memory cell containing the optimal distribution is labelled *Finalized*. The intermediate solution is a solution which may not be optimal. The variable *status* determines the type of the retrieved solution. If no solution has been saved for the node or the parallel execution time of the retrieved solution is greater than or equal to τ (given by the status, *NOT_SOLUTION*), we return from *HPOPTA_Kernel*. If the saved solution in the Mem is the optimal one (given by the status, *SOLUTION*), the retrieved solution is used and we return from *HPOPTA_Kernel*. However, if the retrieved solution is not *Finalized* (given by the status, *SOLUTION_RESUME*), the function *ProcessSolution* is invoked to process this solution (Line 21). Then the function *Backtrack* is invoked (Line 23) to determine whether the routine backtracks or resumes the process from the data point ($x_{c\ idx}$, GETTIME($T_i, x_{c\ idx}$)) where idx has been set by the function *ReadMemory*. If none of the above cases takes place, the routine resumes from data point idx (Line 31).

The $while$ loop (Lines 31-47) scans the time function T_c from left to right examining the data points with execution times less than the time threshold, τ. In each iteration, the data point idx is extracted from the time function T_c. Its problem size $x_{c\ idx}$ is stored in array X_{cur} (Line 32). If this problem size ($x_{c\ idx}$) is equal to n, we found a solution. In this case, the solution is processed using *ProcessSolution()*. Otherwise, if $x_{c\ idx}$ is less than n, *HPOPTA_Kernel* is re-invoked to solve *HPOPT* for the remaining workload $n - x_{c\ idx}$ at the next level L_{c+1} (Lines 33-38). If $x_{c\ idx}$ greater than n, *HPOPTA_Kernel* drops this data point and moves to the next one.

After data point $x_{c\ idx}$ is examined, the function *Backtrack* is called to decide whether the algorithm backtracks or continues the examination at level L_c (Line 40).

Lines 43-46 check if the algorithm reaches the end of the time function T_c. If this is the case, the *while* loop (Line 31-47) terminates and the corresponding memory cell is finalized (Line 48). Otherwise, idx is incremented moving to the next data point in the time function T_c.

Algorithm 2 Algorithm of Recursive Kernel Invoked by Algorithm 1

```
1: function HPOPTA_KERNEL(n, p, c, T, τ, σ, bk, X_cur, Mem, X_opt)
2:     if CUT(n, σ_c) then
3:         return
4:     end if
5:     if c = p − 1 then
6:         if GETTIME(T_c, n) < τ then
7:             x_cur[c] ← n
8:             PROCESSSOLUTION(p, T, τ, σ, bk, X_cur, Mem, −1, X_opt)
9:         end if
10:        return
11:    end if
12:    idx ← −1
13:    if c > 0 ∧ c ≤ p − 2 then
14:        status ← READMEMORY(n, p, c, τ, T, X_cur, Mem, idx)
15:        if status = NOT_SOLUTION then
16:            return
17:        else if status = SOLUTION then
18:            PROCESSSOLUTION(p, T, τ, σ, bk, X_cur, Mem, c, X_opt)
19:            return
20:        else if status = SOLUTION_RESUME then
21:            PROCESSSOLUTION(p, T, τ, σ, bk, X_cur, Mem, c, X_opt)
22:            tempTime ← GETTIME(T_c, x_c idx)
23:            if BACKTRACK(n, c, bk, idx, tempTime, τ, Mem, TRUE) then
24:                return
25:            end if
26:        end if
27:    end if
28:    if idx = −1 then
29:        x_c idx ← 0
30:    end if
31:    while GETTIME(T_c, x_c idx) < τ do
32:        x_cur[c] ← x_c idx
33:        if x_c idx = n then
34:            x_cur[i] ← 0, ∀i ∈ [c + 1, ··· , p − 1]
35:            PROCESSSOLUTION(p, T, τ, σ, bk, X_cur, Mem, −1, X_opt)
36:        else if n > x_c idx then
37:            HPOPTA_KERNEL(n − x_c idx, p, c + 1, T, τ, σ, bk, X_cur, Mem, X_opt)
38:        end if
39:        tempTime ← GETTIME(T_c, x_c idx)
40:        if BACKTRACK(n, c, bk, idx, tempTime, τ, Mem, FALSE) then
41:            return
42:        end if
43:        if idx + 1 = m then
44:            break
45:        end if
46:        idx ← idx + 1
47:    end while
48:    MAKEFINAL(Mem[c][n])
49: end function
```

3.5.2 Theoretical Analysis of *HPOPTA*

Proposition. *The algorithm* HPOPTA *always returns an optimal distribution for a given workload n between p heterogeneous processors which minimizes its parallel execution time.*

Proposition. *The time complexity of* HPOPTA *is $O(m^3 \times p^3)$. The total memory used by the algorithm is $O(p \times (m + n))$.*

The proofs of these two propositions can be found in Appendix B. We will also explain the practical time complexity of *HPOPTA* is enormously less than the theoretical one.

3.6 Experimental Analysis of *HPOPTA*

In this section, we experimentally examine the proposed algorithm, *HPOPTA*. We also present speedup compared to solutions returned by state-of-the-art workload distribution approaches, which are based on functional and constant performance models, and also the straightforward load-balancing algorithm. Two sets of experiments are conducted. The first set is carried out on a real heterogeneous server, while the second is performed on simulated clusters of heterogeneous nodes. Finally, we analyse a hierarchical two-level workload distribution algorithm that uses *HPOPTA* and *POPTA* [3].

3.6.1 Experimental Platform and Applications

We perform our experiments on *HCLServer01* containing an Intel Haswell multicore CPU, Nvidia K40c GPU, and Intel Xeon Phi 3120P, whose specifications are given in Tables 3.1, 3.2 and 3.3, respectively.

We experiment with two widely known scientific data-parallel applications, Matrix Multiplication and 2D discrete Fourier Transform. These applications are configured for execution on *HCLServer01*. Each application consists of three computational kernels running in parallel on three abstract processors of the *HCLServer01*, which are named *CPU*, *GPU*, and *Xeon Phi*.

The Matrix Multiplication application (DGEMM) executes a highly optimized native kernel for CPU and highly optimized out-of-card kernels for the accelerators. The out-of-card kernels allow the GPU and Xeon Phi abstract processors to execute tasks of arbitrary size, not just the ones that fully fit in the accelerator memories. For the multicore CPU, Intel MKL DGEMM [143] is used. For *GPU*, ZZGEMMOOC out-of-card package [144] is used that reuses CUBLAS [145] for in-card DGEMM calls. For Xeon Phi, XeonPhiOOC out-of-card package [146] is used that reuses MKL BLAS [143] for in-card DGEMM calls. In Chapter 6, we will introduce our library facilitating out-of-card computation on accelerators and illustrate the structure of ZZGEMMOOC and XeonPhiOOC out-of-card packages.

The 2D FFT application uses Intel MKL FFT [147] for the multicore CPU and Xeon Phi. For the Nvidia GPU, CUFFT [148] is used. Unlike the Matrix Multiplication application, all computations for FFT are in-card.

The Intel MKL and CUDA versions used are 2017.0.2 and 7.5, respectively. Since the number of threads per core in Intel Haswell is equal to 2, the Intel MKL DGEMM kernel for the multicore CPU uses 44 threads executing on 22 out of 24 physical cores.

3.6.2 Data Partitioning on a Single-node Hybrid Server

In this section, we examine our proposed algorithm on *HCLServer01*. For each application, the input to *HPOPTA* are three time functions representing the performance profiles of the CPU, GPU, and Xeon Phi abstract processors, respectively.

As explained in Section 3.1, the time functions of an application are built simultaneously on all abstract processors to take into account resource contention. It should be mentioned that there is no specific reason for choosing particular problem sizes in our time functions. *HPOPTA* can deal with any time function represented by a discrete set of data points. However, if the consecutive problem sizes are separated by a large step size, the shape of the speed functions becomes smoother thereby disallowing any opportunity for optimization.

We compare the speedup of *HPOPTA* over the state-of-the-art workload distribution algorithms based on functional performance model (FPM), constant performance model (CPM) [14, 110, 15] and the straightforward load-balancing algorithm. As explained before, the state-of-the-art *FPM*-based algorithms suppose a smooth and convex shape for speed functions. Therefore, we build smooth speed functions from the actual ones and use them to obtain *FPM*-based workload distributions. Just for comparison purposes, we will call the *FPM*-based workload distribution approach, *smooth-FPM* algorithms.

The percentage speedup of *HPOPTA* against *smooth-FPM* algorithm is calculated as follows: $Speedup_{FPM}(\%) = \frac{t_{smooth-FPM} - t_{HPOPTA}}{t_{HPOPTA}} \times 100$, where $t_{smooth-FPM}$ and t_{HPOPTA} respectively are the execution times of solutions found by executing *HPOPTA* using smoothed and actual time functions. $t_{smooth-FPM}$ is estimated as follows. First, the workload distribution for a given workload is found by executing *HPOPTA* using smoothed time functions as input. Then, the execution time for this distribution is calculated using the original, not smoothed, time functions. Thus, the smoothed time functions are used for finding the FPM workload distribution, and its execution time is then found using the real time functions.

The percentage speedup of *HPOPTA* against constant performance model is calculated as follows: $Speedup_{cpm}(\%) = \frac{t_{CPM} - t_{HPOPTA}}{t_{HPOPTA}} \times 100$, where t_{CPM} and t_{HPOPTA} respectively are the execution times of solutions found by executing the CPM-based algorithm and *HPOPTA* using actual time functions. The constant performance model (CPM) uses relative speeds of processors, which are constant floating-point numbers. We use three CPMs for comparison and these are determined from three different data points in speed functions of the processors.

We also compare the speedup of *HPOPTA* over the straightforward load-balancing algorithm. In this experiment, the actual functions are utilized to take fluctuations in performance profiles into consideration for finding load-balanced solutions. A load balance solution is one with the minimum difference between the execution times of processors. The number of processors with a non-zero workload (active processors) in load-balanced solutions may be less than the total number of processors. The percentage

speedup of *HPOPTA* against load-balancing algorithm is calculated as follows: $Speedup_{balance}(\%) = \frac{t_{balance} - t_{HPOPTA}}{t_{HPOPTA}} \times 100$, where $t_{balance}$ and t_{HPOPTA} respectively are the execution times of solutions with minimum difference between the execution times of processors, and *HPOPTA* is the execution time of solution found using our proposed algorithm.

We now summarize the experimental results on *HCLServer01* using two data parallel applications Matrix Multiplication and FFT.

Matrix Multiplication

Heterogeneous Matrix Multiplication application uses three kernels to perform computation on CPUs, GPUs and Xeon Phis. It is a data parallel application enabling in-card and out-of-card matrix multiplication on *CPU*, *GPU* and *Xeon Phi*.

The functions which are labelled as *Original* in Figure 3.13 show the speed functions for the three processors (in Floating Point Operations Per Second (FLOPS)). Each speed function of DGEMM (and its equivalent time function) is represented by a discrete set of cardinality (m) equal to 700 data points with problem sizes $x = \{64^2, 128^2, \cdots, 44800^2\}$. Out-of-card DGEMM invocations are performed on GPU and Xeon Phi when workload exceeds the size of main memory on the accelerators. For a problem size n^2 in the speed function, the speed is calculated as $\frac{2 \times n^3}{t}$ where t is execution time taken to multiply two $n \times n$ square matrices. For GPU and Intel Xeon Phi, the execution time includes the transfer of matrices from the host to the device and the results from the device to the host.

To obtain smooth speed function from the actual speed function, we smooth the actual speed function using a polynomial trend line in LibreOffice Calc and construct its equivalent time function. Figures 3.13 shows the original and smoothed speed functions of DGEMM.

From the figure, we can observe the following:

- Xeon Phi speed function is almost smooth between 64^2 to 13760^2. However, the variations increase for larger problem sizes (13824^2 and beyond) where DGEMM out-of-card computations are invoked. Unlike

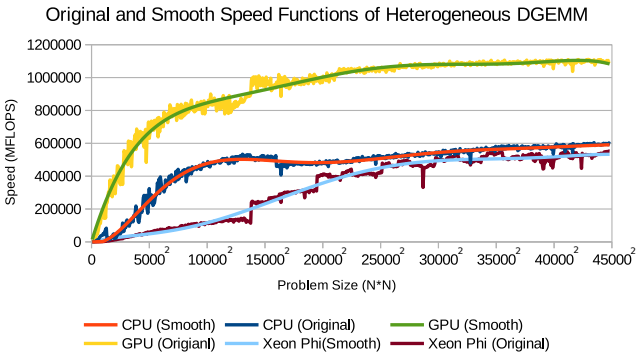

Figure 3.13: Original and smoothed speed functions of the heterogeneous Matrix Multiplication on HCLServer01*. MKL DGEMM is invoked for CPU and Xeon Phi. For GPU, CUBLAS is used. The original functions are smoothed using polynomial trend line in LibreOffice Calc.*

Xeon Phi, the variations decrease for CPU and GPU as problem size increases. The maximum variations for CPU, GPU and Xeon Phi are 700%, 50% and 150%, respectively. The maximum variations for Xeon Phi occur for problem sizes in the range $[12800^2, 19200^2]$.

- The shapes violate the assumptions of FPMs. Therefore, load-balancing data partitioning algorithms based on FPMs may not return optimal solutions.
- The new model-based methods proposed in [16], [3] cannot be used since they assume all the available processors to be identical and therefore take a single speed function as an input.

To determine the percentage improvements given by *HPOPTA*, we create an experimental data set for DGEMM whose data points ranges from $(\frac{p}{3} \times 64 \times 100)^2$ to $(p \times 64 \times 700)^2$ with step size of 64^2. Since there are three abstract processors in the *HCLServer01*, p is equal to 3 in this experiment. Figure 3.14 shows the speed of heterogeneous DGEMM on *HCLServer01* when executed using *HPOPTA* in comparison with *FPM* workload distribution. *HPOPTA* gives the minimum, average, and maximum percentage of improvement of 0, 14, 261 percent respectively in comparison with *FPM*. Since $t_{smooth-FPM}$ equals t_{HPOPTA} for some workloads, we have observed zero percentage of

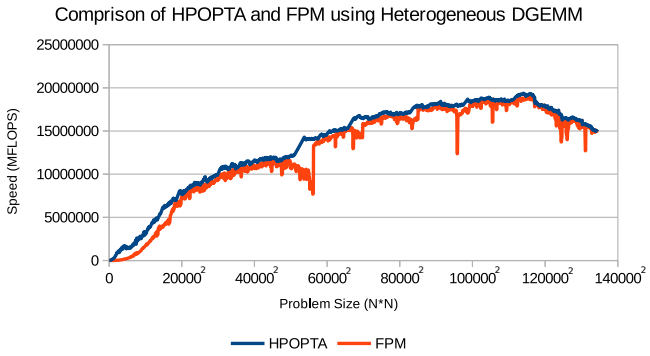

Figure 3.14: Speed functions of the heterogeneous Matrix Multiplication for whole HCLServer01*. The application is executed for each problem size* n *using two different workload distributions* HPOPTA *and* FPM*.*

improvement for them. The maximum improvement belongs to the workload 6784^2 where the problem sizes for CPU, GPU and Xeon Phi found using original functions (*HPOPTA*) are $\{1856, 4928, 0\}$ and using the smooth functions are $\{576, 4736, 1472\}$.

For *CPM*, we use the relative speeds of the processors based on the execution of one problem size. We select three different problem sizes from the speed functions for this purpose. One at the beginning, one in the middle, and one in the end. These are 4736, 28672 and 44800 and therefore there are three constant relative performance models, $\{0.34, 0.59, 0.07\}$, $\{0.27, 0.55, 0.18\}$, $\{0.27, 0.49, 0.24\}$ where the first element in each set is relative speed of CPU, the second is relative speed of GPU, and the last one represents the relative speed of Xeon Phi. The average percentage improvements are respectively 122, 106, and 82 percent. Since the number of data points in speed functions is limited, there are workload whose CPM workload distributions contain problem sizes, which exceed the largest problem size in the speed functions. That is, for these workload, CPM-based algorithm does not find any solution. Therefore, we ignored these workload sizes to calculate the maximum and average of $Speedup_{cpm}$.

We use the aforementioned experimental data set to compare *HPOPTA* workload distribution against the straightforward *load-balancing* approach. Figure 3.15 shows the speed of heterogeneous DGEMM on *HCLServer01*

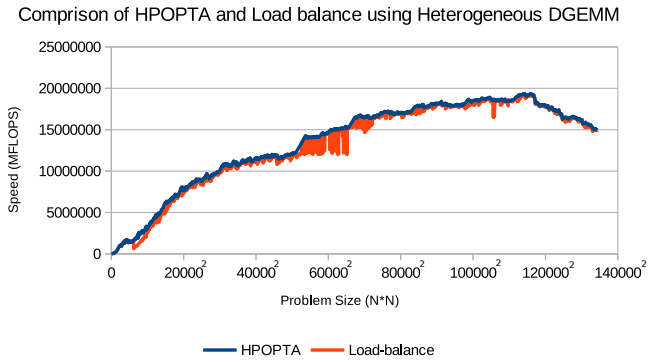

Figure 3.15: Speed functions of the heterogeneous Matrix Multiplication for whole HCLServer01*. The application is executed for each problem size* n *using two different workload distributions* HPOPTA *and* load-balancing*.*

when executed using *HPOPTA* in comparison with load-balanced workload distribution. *HPOPTA* gives the minimum, average, and maximum percentage of improvement of 0, 5, 143 percent respectively in comparison with *load-balancing*. Since $t_{balance}$ equals t_{HPOPTA} for some workloads, we have observed zero percentage of improvement for them.

FFT

Heterogeneous FFT application uses three kernels to perform computation on CPUs, GPUs, and Xeon Phis. It is a data parallel application enabling in-card fast Fourier transform on the three abstract processors in *HCLServer01*.

The FFT speed functions are shown in the Figure 3.16 (the functions which are labelled as *Original*). The discrete set for the FFT speed functions has the cardinality 1090 and contains problem sizes, $\{16^2, 32^2, \cdots, 24000^2\}$. For a problem size n^2 in the speed function, the speed is calculated as $\frac{n^2 \times \log_2 n^2}{t}$ where t is execution time taken to compute 2D FFT of size n^2. It does not include problem sizes, which cannot be factored into primes less than or equal to 127. For these problem sizes, CUFFT for GPU gives failures. Unlike DGEMM, all the FFT invocations are performed in-card.

Figures 3.16 shows the original and smoothed speed functions of FFT. We again apply polynomial trend line in LibreOffice Calc on actual speed function

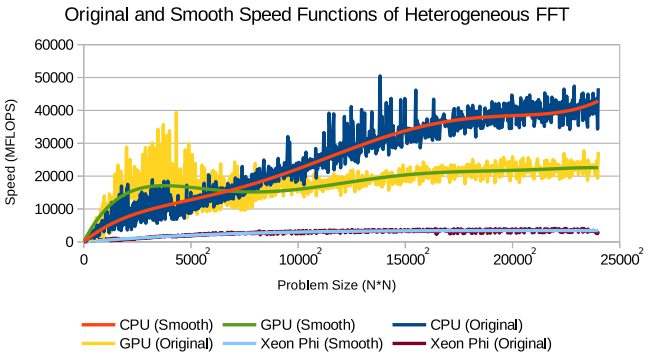

Figure 3.16: Original and smoothed speed functions of the heterogeneous FFT application on HCLServer01*. MKL FFT is invoked for CPU and Xeon Phi. For GPU, CUFFT is used. The original functions are smoothed using polynomial trend line in LibreOffice Calc.*

of FFT to obtain its smooth function.

From the figure, we can observe that Xeon Phi is markedly slower than CPU and GPU. It is because the execution time of communications between Xeon Phi and host CPU dominates the execution time of computations performed by Xeon Phi. However, GPU uses optimized data transfers by deploying two data engines (for transfers from CPU host to GPU and from GPU to CPU host) and does not suffer from this problem. The maximum variations for CPU, GPU, and Xeon Phi are almost 350%, 560% and 200%, respectively. The maximum variations for Xeon Phi occur for problem sizes in the range of $[16^2, 800^2]$. It can be seen again that the shapes violate the assumptions on shape of FPMs. Therefore, load-balancing data partitioning algorithms based on FPMs may not return optimal solutions. Also the new model-based methods proposed in [16], [3] cannot be used for this case.

To analyse FFT, the experimental data set includes data points ranging from $(\frac{p}{3} \times 16 \times 100)^2$ to $(p \times 16 \times 1500)^2$ with step size of 16^2 ($p = 3$ for *HCLServer01*). Figure 3.17 compares the speed of heterogeneous FFT when executed using *HPOPTA* with the speed when the workload is distributed using *FPM*. *HPOPTA* gives the minimum, average, and maximum percentage of improvements of 0, 40, and 502 percent respectively in comparison with *FPM*. The maximum improvement happens for the workload, 1920^2. The problem

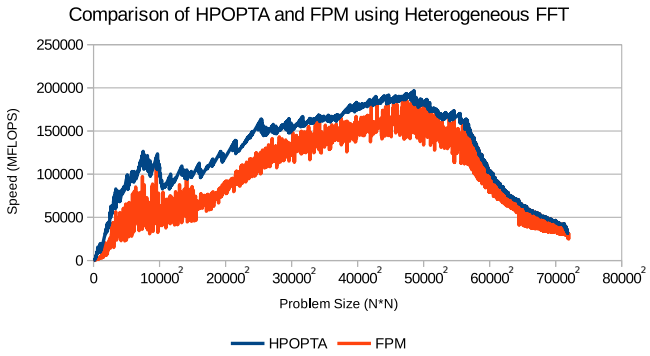

Figure 3.17: Speed functions of the heterogeneous FFT for whole HCLServer01*. The application is executed for each problem size n using two different workload distributions* HPOPTA *and* FPM*.*

sizes given to CPU, GPU and Xeon Phi using original functions (*HPOPTA*) are $\{464, 1456, 0\}$ and using the smooth functions are $\{656, 1168, 96\}$.

Like DGEMM, to compare *CPM*-based algorithm with *HPOPTA*, we use the relative speeds based on three different problem sizes, 4320, 13824, and 24000, from the speed functions. These points result in three CPMs, $\{0.18, 0.78, 0.04\}$, $\{0.69, 0.26, 0.05\}$, $\{0.60, 0.35, 0.05\}$. The average percentage improvements are 301, 164, and 129 percent respectively. Since the number of data points in speed functions is limited, there are workload whose CPM workload distributions contain problem sizes, which exceed the largest problem size in the speed functions. In addition to out-of-range problem sizes, there is no speed for problem sizes, which cannot be factored into primes less than or equal to 127. This is due to failure of CUFFT calls for these problem sizes. That is, for these workload sizes, CPM-based algorithm does not find any solution. Therefore, we ignored these workload sizes to calculate the maximum and average of $Speedup_{cpm}$.

We use the aforementioned experimental data set to compare *HPOPTA* workload distribution against *load-balancing* approach. Figure 3.18 shows the speed of heterogeneous FFT on *HCLServer01* when executed using *HPOPTA* in comparison with the straightforward load-balanced workload distribution. *HPOPTA* gives the minimum, average, and maximum percentage of improvement of 0, 19, 331 percent respectively in comparison with *load-balancing*.

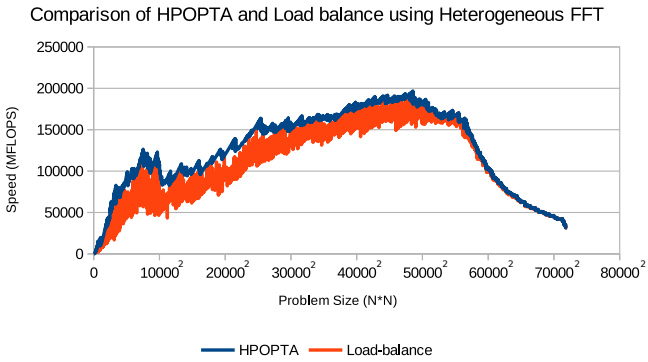

Figure 3.18: Speed functions of the heterogeneous FFT for whole HCLServer01*. The application is executed for each problem size n using two different workload distributions* HPOPTA *and* load-balancing*.*

Discussion

We observed a tight correlation between the average variations in speed functions and the average performance improvements. To study this correlation further, we create speed bands for DGEMM and FFT speed functions as mentioned in [149]. By looking at DGEMM speed functions in Figures 3.13 and 3.16, it can be observed that there are maximum differences of 29% and 150% approximately between lower and upper bands in DGEMM and FFT speed functions. These differences confirm the achieved improvements where the average $Speedup_{FPM}$ of FFT is about four times greater than that of DGEMM.

The experimental results revealed that applying load-balancing data partitioning algorithms, either considering variations (straightforward load-balancing algorithm) or not (*CPM* and *FPM*), may not return optimal solutions on modern hybrid platforms. This is because of complex shapes of performance profiles on these systems.

We also observed that sometimes the number of processors in the optimal solutions determined by *HPOPTA* is less than p. For instance, the optimal solution for FFT for matrix size 1200×1200 uses just one abstract processor, GPU, meanwhile, for matrix size 19632×19632 the optimal distribution only uses CPU and GPU.

3.6.3 Using *HPOPTA* for Data partitioning on Clusters of Heterogeneous Nodes

In this section, we describe how *HPOPTA* can be used to optimally distribute workload between processors in a cluster of heterogeneous nodes. We will also present a hierarchical two-level workload distribution approach based on *HPOPTA* and *POPTA* [3], which not only reduces the computational complexity but also allows parallel computation for finding optimal workload distribution. *POPTA* is an algorithm for performance optimization on *homogeneous* platforms using functional performance models. We would like to mention that the incorporation of the cost of communications is out of the scope of this dissertation.

To study the performance improvements given by *HPOPTA* at scale, we simulated clusters consisting of $8, 16, \cdots, 256$ *HCLServer01* nodes, where each node has three abstract processors and therefore the total number of heterogeneous abstract processors ranges from 24 to 768. We conduct experiments that are a combination of actual measurements conducted on *HCLServer01* and simulations for clusters containing replicas of *HCLServer01*. The actual measurements include the construction of time functions, which are input to *HPOPTA* (refer Section 3.6.2). The simulations contain the execution of *HPOPTA* to determine workload distributions, which allow us to calculate the parallel execution times of computations in the data-parallel applications and consequently the speedups demonstrated by *HPOPTA*.

HPOPTA requires as input, the time function of each abstract processor in the simulated cluster. Since all nodes are identical, we build the time functions for one node, *HCLServer01*, and then use them for all nodes in the simulated cluster. For example, for a simulated cluster consisting of 8 *HCLServer01* nodes, the input to *HPOPTA* will consist of 24 (3×8) time functions.

We now examine *HPOPTA* on simulated clusters of heterogeneous nodes using the same data parallel applications, Matrix Multiplication and FFT.

Matrix Multiplication

For each simulated cluster, we execute DGEMM using a test data set whose data points ranges from $(\frac{p}{3} \times 64 \times 100)^2$ to $(p \times 64 \times 700)^2$ with step size of 64^2. The obtained results show that *HPOPTA* gives the minimum, average, and maximum percentage improvements of 0, 14, and 261 percent respectively in comparison with *FPM*.

We choose the same problem sizes 4736, 28672, and 44800 to obtain relative speeds for *CPM* workload distribution. The average percentage improvements of *HPOPTA* over *CPM* are 122, 106, and 82 percent, respectively.

FFT

To analyse FFT, the experimental data set include data points ranging from $(\frac{p}{3} \times 16 \times 100)^2$ to $(p \times 16 \times 1500)^2$ with step size of 16^2. The obtained results show *HPOPTA* gives the minimum, average and maximum percentage of improvement of 0, 43, 513 percent respectively in comparison with *FPM*.

We choose the same problem sizes 4320, 13824, and 24000 to obtain relative speeds for *CPM* workload distribution. The average percentage of improvement of *HPOPTA* over *CPM* are 301, 164 and 129 percent, respectively.

Discussion

We observed almost the same percentage of improvement for different cluster sizes for both DGEMM and FFT. It can be concluded that the performance improvement is independent of p assuming the cost of communications is not taken into account.

There is a strong correlation between average performance improvements and the average variations in speed functions. Furthermore, the maximum performance improvement over FPM cannot exceed the maximum variation in the speed functions. In our experiments, all nodes in simulated clusters are identical and their speed functions consequently will be identical. Thus, average and maximum performance improvements of a simulated cluster consisting of identical nodes are not related to the number of nodes but related to

the shapes of speed functions which are identical for all nodes.

We would like to mention that the study and incorporation of communication costs is a significant body of work and is therefore out of scope of this thesis. It is the focus of our current research.

In addition, the number of abstract processors in the optimal solution determined by *HPOPTA* is often less than p. For example, in a cluster of eight *HCLServer01* nodes, the optimal solution for FFT for matrix size 304×304 uses just one GPU while the other 23 abstract processors are given zero problem size. For FFT for matrix size 25552×25552 the optimal workload distribution uses 21 abstract processors leaving one CPU and two Xeon Phis unused.

3.6.4 Hierarchical Two-level Workload Distribution

In Section 3.6.3, we used *HPOPTA* for optimal workload distribution in a cluster of identical hybrid nodes. As *HPOPTA* is oblivious of the regular structure of the underlying platform, in order to find an optimal solution for a cluster of h *HCLServer01* nodes it had to analyse $3 \times h$ time functions. In this section, we present an hierarchical workload distribution algorithm, *HiPOPTA*, which combines *HPOPTA* and *POPTA* [3] to find an optimal solution for a cluster of h identical nodes only using $c + 1$ time functions instead of $c \times h$, where c is the number of heterogeneous processors in one node.

HiPOPTA first distributes workload between identical nodes (Inter-node workload distribution) using *POPTA*. The input to it is a whole speed function of a node constructed using *HPOPTA*. The assigned problem size to each node is then distributed between the processing elements of each node (Intra-node workload distribution) using *HPOPTA*.

We explain the steps of *HiPOPTA* using a cluster of *HCLServer01* nodes:

- **Building speed function of whole *HCLServer01* using *HPOPTA*:** For each workload, we run the heterogeneous application on *HCLServer01* using the *HPOPTA* workload distribution and measure its parallel execution time. The resulting speed function characterizes the performance of *HCLServer01* as a whole. Since all the nodes in the simulated cluster are identical, their speed functions will be the same, too. Figures 3.14

and 3.17 respectively show the speed functions of Matrix Multiplication and 2D FFT of the whole *HCLServer01*. In Section 3.6.2, we have explained in detail how these speed functions are built for *HCLServer01*.

- **Inter-node workload distribution:** We use the whole *HCLServer01* speed function to distribute workload between the nodes of the simulated cluster. Since all nodes are identical, we can use *POPTA* [3] for finding the optimal workload distribution between nodes.
- **Intra-node workload distribution:** *HPOPTA* is then applied inside each node to divide the assigned workload between CPU, GPU, and Xeon Phi of this node so that the execution time is minimized. The intra-node workload distributions can be determined by running *HPOPTA* on the nodes of the cluster in parallel.

To evaluate *HiPOPTA*, we repeat experiments conducted in Section 3.6.3 with the same experimental data sets. As expected, the resulting execution times of the distributions returned by *HiPOPTA* are the same as the ones obtained in the section 3.6.3 using plain *HPOPTA*.

The reason behind the use of two-level partitioning is the reduction in theoretical and practical complexities for finding optimal distributions on large scale clusters. Assume a cluster involving h identical nodes where each node consists of c processors. Therefore, the cluster totally comprises $c \times h$ processors ($p = c \times h$). Let cardinality of time functions be m where $c >> m$ and $h >> m$. We first calculate the time complexity of *HiPOPTA*. There are h identical nodes and therefore *POPTA* finds the optimal inter-node distribution with the time complexity of $O(h^2)$ [3]. Optimal intra-node workload distributions are then found using parallel executions of *HPOPTA* on h nodes with time complexity of $O(c^3)$. Therefore, the total theoretical complexity will be equal to $O(h^2 + c^3)$. The theoretical complexity of the non-hierarchical partitioning is $O(p^3)$, which is equal to $O(c^3 \times h^3)$. Therefore, *HiPOPTA* is $O(\frac{c^3 \times h^3}{h^2 + c^3})$ times faster than the non-hierarchical one. In addition, the hierarchical workload distribution allows parallel computations to find optimal distribution.

HiPOPTA always returns an optimal distribution. Indeed, according to [3], *POPTA* finds an optimal workload distribution between identical compute

nodes represented by their speed function. Assuming that the speed function of a node reflects the fastest speed of execution of any given workload, it will find a globally optimal distribution. However, by construction, the speed function of a node as a whole found locally by *HPOPTA* does give the fastest possible speed of execution for any workload given to the node. Note that since there may be more than one optimal distribution, distributions returned by *HiPOPTA* and *HPOPTA* may be different. However, their execution times will be always the same.

3.7 Summary

Modern high-performance computing platforms have become highly heterogeneous due to the tight integration of multicore CPU processors and accelerators. This tight integration causes contention on shared resources such as Last Level Cache (LLC), DRAM, PCI-E links, etc. Due to this serious contention and NUMA, the performance profiles of data-parallel applications executing on these heterogeneous platforms are not smooth and deviate greatly from the shapes that supposed by state-of-the-art load-balancing algorithms to find optimal solutions.

In this chapter, we formulated the problem of finding optimal distribution on heterogeneous clusters of hybrid nodes and proposed a novel model-based data partitioning algorithm to minimize the execution time for general performance profiles of data-parallel applications executing on clusters of heterogeneous nodes. The inputs to the algorithm are the problem size, n, p discrete time functions, which represent the performance profiles of p processors existing in the heterogeneous cluster. The time complexity of the proposed algorithm is $O(m^3 \times p^3)$ where m and p respectively represent the maximum cardinality of input time function and the number of heterogeneous processors. We studied the optimality of solutions found by the proposed algorithm using two well-known data-parallel applications, matrix multiplication and two-dimensional discrete fast Fourier transform. According to the experimental results, the proposed algorithm demonstrated considerable improvements in

average and maximum performance for the two applications in comparison with state-of-the-art load-balancing algorithms.

The software implementation for *HPOPTA* is available at [150].

In our future work, we aim to design and implement parallel versions of the algorithms to reduce the theoretical complexity and develop extensions to *HPOPTA* for optimization of applications with 2D and 3D problem dimensions. We would also extend the proposed algorithm to consider the cost of communications.

Chapter 4

A Novel Model-based Algorithm for Dynamic Energy Consumption Optimization of Data-Parallel Applications on Heterogeneous HPC Platforms

Energy Consumption is one of the main challenges hindering High-Performance Computing community from breaking exascale barrier [151]. Current HPC systems are already consuming Megawatts of energy. For example, the world's most powerful supercomputer as of 2018, Summit, consumes around 10 Megawatts of power (including the cooling power) [8], and US Department Of Energy (DOE) aims to deploy an exascale supercomputer, capable of performing exa-FLOPS (10^{18}) in a power envelope of 20-30 megawatts [152]. Because of such high power consumption, future HPC systems are highly likely to be power constrained.

As explained in Chapter 1, energy profiles of applications executing on uniprocessors are nicely linear or smooth [31, 30, 3, 18], where a vast majority of algorithms solving energy optimization problem deploy analytical modelling to estimate the energy consumption of applications and then adjust DVFS levels

and the number of cores/threads to achieve higher energy efficiency [31, 32, 33, 47, 48, 49]. These optimization algorithms consider a linear relationship between workload size and energy consumption.

Emerging multi-core processors and also their tight integration with many-core accelerators have incurred new complexities, such as resource contention and NUMA, which lead to complicated nodal architectures. These complexities pose new challenges to energy optimization:

- There is a complex relationship between workload size and energy consumption of applications executing on modern heterogeneous platformers where the shape of energy profiles may be non-linear and even non-convex with variations,
- Analytical models, which ignore workload size to estimate energy consumption, have been reported to be inaccurate where cannot reveal the exact real-life behaviour of parallel applications on hybrid HPC platforms [53, 52, 54, 55].

Therefore, workload distribution has now become an important decision variable that cannot be ignored in solving the energy optimization problem. Workload distribution has been already taken into account as the only decision variable for dynamic energy optimization on *homogeneous* clusters [3, 18]. Nevertheless, to the best of our knowledge, there is no variation-aware approach considering workload partitioning as a decision variable for dynamic energy optimization on *heterogeneous* HPC platforms.

In this section, we formulate the problem of optimization of data-parallel applications on heterogeneous HPC platforms for dynamic energy through *workload distribution* and propose a model-based data-partitioning algorithm, which is named *HEOPTA* (**H**eterogeneous dynamic **E**nergy **OPT**imization **A**lgorithm), to solve the problem. This algorithm minimizes the dynamic energy consumption of data-parallel applications, running on heterogeneous platforms, for the most general shapes of dynamic energy profiles by determining optimal workload distributions.

Another significant challenge is *energy modelling* of heterogeneous parallel application running on hybrid platforms. Consider a parallel application

running on a CPU, a GPU and an Xeon Phi. *HEOPTA* requires the dynamic energy profile of each computational kernel separately, which is referred to as the fine-grained decomposition of the energy consumption in hybrid platforms.

The dynamic energy consumption modelling of heterogeneous HPC platforms, including tightly integrated resources, in a granularity of component-level is a challenging task. To address this challenge, we will propose an *additivity* approach, based on the notion of loosely-coupled abstract processors, explained in Chapter 3, and system-level energy measurements. Using this approach, we can build the individual discrete dynamic energy function of each abstract processor within sufficient accuracy for the optimization algorithm to improve the average energy efficiency. Although restricted by some limitations (such as fitting workloads in the main memory to avoid paging on disks), the proposed solution is a simple and practical methodology to accurately enough determine the dynamic energy consumption of every single computational kernel in parallel applications. The methodology is purely based on physical measurements using external power meters.

We experimentally analyse the accuracy of our energy modelling methodology and the efficiency of *HEOPTA* using two data-parallel applications, matrix multiplication and 2D fast Fourier transform, on a cluster of two heterogeneous nodes.

4.1 Terminology

We explain here different terms used in this thesis related to energy consumption.

The power consumption of hardware components in a computing system can be classified into: a). dynamic power, and b). static power. From an application point of view, dynamic and static power consumption are defined as the power consumed by the whole system with and without the given application execution, respectively.

Two types of energy consumption can consequently be considered: a). static energy, and b). dynamic energy. Static energy consumption is equal to

the energy consumed by the platform without the application execution. The dynamic energy consumed by executing an application is calculated by subtracting this static energy consumption from the total energy consumption of the platform.

If P_S represents the static power consumption of a platform, E_T is the total energy consumption of the platform during the execution of an application, with an execution time of T_E seconds, then the dynamic energy E_D can be determined as:

$$E_D = E_T - (P_S \times T_E) \tag{4.1}$$

4.2 Dynamic Energy Measurement in Heterogeneous Platforms

In this section, we explain how to model the dynamic energy consumption of parallel applications using a set of discrete dynamic energy functions. Each function consists of a set of data points where each point represents the application dynamic energy consumption running on a processor for a given problem size. Like speed functions (Chapter 3), energy functions are application-specific and built empirically.

Building energy functions in modern heterogeneous platforms requires accurate measuring the dynamic energy consumption of each computational kernel in a hybrid application. To address this challenge, we propose a new methodology which provides a fine-grained decomposition of dynamic energy consumption for parallel applications via physical measurements using external power meters.

In following sections, we will explain approaches generally used to measure energy consumption in computing systems and then introduce our measuring methodology for heterogeneous platforms using power meters.

4.2.1 Energy Measurement in Computing Platforms

There are two prevalent approaches to measure energy consumption during an application execution:

1. **Physical measurement:** Measuring energy using external power meters or on-chip power sensors,
2. **Software measurement:** Measuring energy using energy predictive models.

Physical measurement is considered to be accurate at the node level, however, it can only provide the measurement at a node level, and therefore, lacks the ability to provide the fine-grained energy consumption of an application or intra-node power consumption details. Consider a heterogeneous server which is facilitated with one power meter sitting between the wall A/C outlet and the input power socket of the server. Although we can accurately measure the energy consumption of the whole node (CPU, accelerators, and the other shared resources contributing in a computation) for executing a hybrid application using the power meter, it cannot determine how much energy is consumed by CPU and each accelerator individually.

Another way for physical measurement of energy is to utilize on-chip power sensors. But not all components of a given system, contributing in the execution of an application, are equipped with power sensors. To illustrate this, consider a hybrid application running in parallel on a hybrid node which includes a CPU and some accelerators. Apart from processing elements, any other components of the given platform, which involve in the application execution, consume energy, but they do not have any power sensor. This way, we cannot measure the amount of energy consumed by most of these components, such as data banks of DRAM or the PCI-E links offloading application data to and from host CPU cores to accelerators and etc. Therefore, using power sensors cannot provide the amount of energy consumption in a fine-grained granularity.

In summary, the straightforward use of physical measurement techniques cannot provide the fine-grained energy consumptions on hybrid nodes.

Software measurement approaches, which rely on energy predictive models, can be considered as an alternative to physical measurements. Performance Monitoring Counters (PMCs) are predominantly employed by a vast majority of these energy predictive models to estimate the energy consumption during an application execution. PMCs are special purpose registers available in modern computing architectures to store software and hardware activities counts. These approaches [153, 32, 154, 41, 155, 39, 119, 44, 122] typically use linear regression of the performance events to model the energy consumption of hardware components (such as CPU, DRAM, disks, fans etc.).

Although energy predictive models have got popular, their prediction accuracy has been put under criticism by some research works [153, 53, 55, 52]. For instance, McCullough et al. [53] showed that the power prediction error of linear-based power modelling approaches can reach as high as 150 percent in modern computing platforms, and that is why they propose direct measurement as an alternative to model-based techniques to deal with the inherent complexities caused by modern architectures.

To summarize, we can straightforwardly use neither existing physical nor software approaches to measure the dynamic energy consumption of hybrid nodes at fine-grained granularity within sufficient accuracy. In Section 4.2.2, we will address this shortage by introducing a novel methodology to determine the fine-grained dynamic energy consumptions of computational kernels, executing in parallel on heterogeneous HPC platforms, within sufficient accuracy. The proposed methodology is based on physical measurements using power meters.

4.2.2 Dynamic Energy Measurement in Hybrid Heterogeneous Platforms

In Section 3.1, we explained how to measure the execution speed for each kernel of parallel applications running on heterogeneous platforms.

While the execution time of each kernel in parallel applications can be measured accurately using high precision timers (processor clocks), there is no such straightforward way to accurately measure the amount of dynamic en-

ergy consumed by each kernel in a hybrid application separately, as explained in Section 4.2.1.

In this section, we, first, outline challenges to measuring energy consumption in fine-grained granularity on heterogeneous platforms and then propose a novel methodology to address this issue using physical energy measurements within sufficient accuracy.

Modern HPC platforms have become highly heterogeneous, and this has posed serious challenges to fine-grained energy measurement of hybrid parallel applications, which are listed below:

1. **Heterogeneity:** Heterogeneous platforms involve different types of processing elements with different computational capabilities and energy consumption. In addition, unlike homogeneous systems, there are different computational kernels running on these processors and accelerators in parallel. It implies that these computational kernels are consuming a different amount of dynamic energy, and hence, their energy consumption should be determined individually, for optimization purposes. However, as explained in Section 4.2.1, this heterogeneity has put real challenges to measuring the amount of energy consumed by each kernel in finer granularity.

2. **Integrity:** Tight integration of CPUs with accelerators incurs some complexities such as resource contention on shared resources and NUMA. Therefore, it becomes difficult to measure the dynamic energy consumption of a computational kernel without considering the impact of other kernels that are running in parallel. Furthermore, unlike the single-core processor's era, these complexities have made analytical models inaccurate to estimate the dynamic energy consumption of applications using a few architectural and program parameters [153, 53, 55, 52], as explained in Section 4.2.1. Thus, employing these models may result in sub-optimal solutions for energy optimization.

To the best of our knowledge, apart from the efforts of our research group in Heterogeneous Computing Lab (HCL), no contemporary approach measures

the *dynamic* energy consumption of such a heterogeneous platform at fined-grained granularity. Now, we elucidate the details of our methodology.

Abstract Processors in Fine-grained Dynamic Energy Measurement

To cope with these challenges, we reuse the notion of *abstract processors* (Section 3.1) to sort processing resources, which contribute into an application execution, into loosely-coupled groups in such a way that we are able to measure their dynamic energy consumption with the accuracy which is sufficient for successful application of the optimization algorithm, proposed later in this chapter. An abstract processor contains only those computing elements which execute a single application kernel.

Figure 4.1 highlights abstract processors for modelling dynamic energy consumption on *HCLServer01*. The hybrid node consists of an Intel multicore CPU connected to one Nvidia GPU and one Intel Xeon Phi, that their specifications have been explained in Chapter 3. The server has been equipped with a Watts Up Pro power meter to measure the energy consumption of the whole node physically.

In Section 3.1, we explained that this node can be classified into three abstract processors, such as: *CPU*, *GPU*, and *Xeon Phi*. The CPU abstract processor involves 22 CPU cores executing the multi-threaded CPU kernel which is highlighted in dark blue. The *GPU* abstract processor containing Nvidia K40c GPU along with its dedicated host CPU core executing the GPU kernel and its PCI-E link which is highlighted in orange. The *Xeon Phi* abstract processor consisting of Intel Xeon Phi 3120P coprocessor along with its dedicated host CPU core executing the Xeon Phi kernel and its PCI-E link.

Apart from computing elements, there are other resources such as Network Interface Card (NIC), Solid State Drive (SSD), fans, chipsets and etc., which are almost shared between all abstract processors and consume energy during an application execution. To eliminate their potential contribution in the dynamic energy consumption of a given abstract processor, we take several precautions in energy measurements, which are listed below:

- Since fans consume a significant and variable amount of energy during

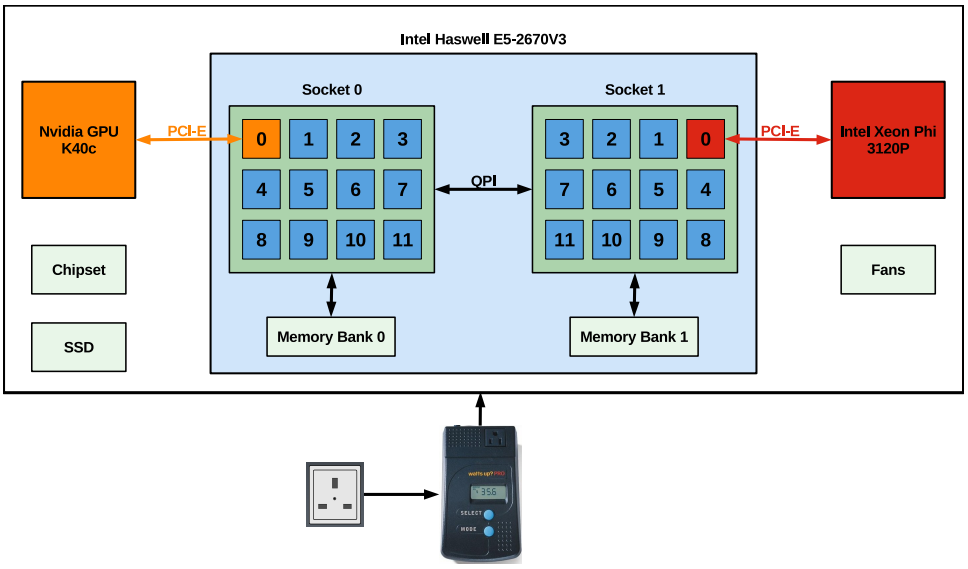

Figure 4.1: Block diagram of HCLServer01 *including an Intel Haswell multi-core CPU, one Nvidia K40c GPU and one Intel Xeon Phi 3120P highlighting abstract processors for modelling dynamic energy consumption. The server is equipped with a Watts Up Pro power meter to measure energy consumption physically.*

an application execution, to eliminate their contribution, we set them at full speed before running the application. Thus, they run consistently at the same speed and consume the same amount of energy which is then considered part of the static energy of the platform. This way, the dynamic energy consumption of a given abstract processor does not rely on fans.

- Disk utilization is monitored during the application run to ensure that it does not use these components. We ensure that the problem size used in the execution of an application does not exceed the main memory, where swapping (paging) does not occur. That is, problem sizes are bounded by main memory size.

- NIC is also monitored. We assure that network is not used by the application. In should be mentioned that communication cost is out of the scope of this research.

Additivity in Fine-grained Dynamic Energy Measurement

We need the dynamic energy consumption of each abstract processor to build dynamic energy functions of a hybrid application. However, as explained earlier, there is no fine-grained approach measuring dynamic energy consumption using physical measurements.

To address this issue, we classified hardware resources into some abstract processors. Now, we are going to use the notion of *additivity* for measuring the dynamic energy consumption of each abstract processor separately. The additivity criterion is based on a simple and intuitive rule, which is defined as *"the amount of dynamic energy consumed by all application kernels running the same workload n in parallel on given p abstract processors is equal to the sum of dynamic energies consumed by these kernels when are separately executed on the same abstract processors with the same workload n"*.

For further illustration, consider a hybrid application executing a workload n on *HCLServer01*. The application consists of three abstract processors, *CPU*, *GPU* and *Xeon Phi*. Suppose $E_{CGX}(n)$ represents the dynamic energy consumed by the parallel running of these three abstract processors. Then, we run each computational kernel separately, to execute the same workload, where their dynamic energy consumptions are respectively $E_{CPU}(n)$, $E_{GPU}(n)$ and $E_{PHI}(n)$. Regarding the additivity notion, parallel dynamic energy consumption ($E_{CGX}(n)$) and combined dynamic energy consumption ($E_{CPU}(n) + E_{GPU}(n) + E_{PHI}(n)$) should satisfy Eq. 4.2.

$$E_{CGX}(n) = E_{CPU}(n) + E_{GPU}(n) + E_{PHI}(n) \tag{4.2}$$

Using additivity, we can build the energy model of a hybrid application running on *HCLServer01* by building energy functions for the abstract processors *CPU*, *GPU* and *Xeon Phi* separately.

In general, we need $p \times m$ experiments to build dynamic energy functions of a hybrid application, within sufficient accuracy, executing on p abstract processors where each discrete function consist of a set of points of cardinality m.

In Section 4.6, we will explain how to practically build energy functions

using Watts Up power meter and then experimentally validate the accuracy of our methodology on two modern heterogeneous hybrid servers.

4.3 Formulation of Heterogeneous Dynamic Energy Optimization Problem

Suppose there exists a problem of size n running on p heterogeneous processors, with discrete dynamic energy functions $E = \{e_0(x), ..., e_{p-1}(x)\}$ where $e_i(x)$, $i \in \{0, 1, \cdots, p-1\}$, is a discrete dynamic energy function of processor P_i with a cardinality of m. Without loss of generality, we assume $x \in \{1, 2, \cdots, m\}$. The heterogeneous dynamic energy optimization problem can be formulated as follows:

Heterogeneous Dynamic Energy Optimization Problem, *HEOPT*(n, p, m, E, X_{opt}, e_{opt}): The problem is to find a workload distribution, $X_{opt} = \{x_0, ..., x_{p-1}\}$, for the workload n running on p heterogeneous processors so that the solution minimizes dynamic energy consumption for the parallel execution of n. The parameters (n, p, m, E) are the inputs to the problem. The outputs are X_{opt}, which is the optimal solution (workload distribution), and e_{opt}, which represents the dynamic energy consumption of the optimal solution. Eq. 4.3 formulates the problem as an INLP problem.

$$
\begin{aligned}
& e_{opt} = \min_{X} \sum_{i=0}^{p-1} e_i(x_i) \\
& \text{Subject to} \quad \sum_{i=0}^{p-1} x_i = n, \\
& \text{where} \quad p, m, n \in \mathbb{Z}_{>0} \quad \text{and} \quad x_i \in \mathbb{Z}_{\geq 0} \quad \text{and} \\
& \qquad\qquad e_i(x) \in \mathbb{R}_{>0}
\end{aligned}
\tag{4.3}
$$

The objective function in Eq. 4.3 is a function of workload distribution X, $X = \{x_0, ..., x_{p-1}\}$, for a given workload n executing on the p processors. The function returns the amount of dynamic energy which is consumed by running

each given distribution X on processors $\{P_0, \cdots, P_{p-1}\}$. The total dynamic energy consumption of X is calculated as the summation of all dynamic energies consumed by the p processors $\{P_0, \cdots, P_{p-1}\}$ which run X in parallel. The distribution with minimum dynamic energy consumption is returned as the optimal distribution. It should be noted that the number of active processors (processors with non-zero workload sizes) in the optimal solution determined by *HEOPTA* (X_{opt}) might be less than p.

4.4 *HEOPTA*: Algorithm Solving HEOPT Problem

In this section, we will introduce *HEOPTA*, an efficient branch-and-bound algorithm solving the aforementioned dynamic energy optimization problem, *HEOPT*. Before exploring the candidate solutions of a branch, the branch is checked against two upper estimated bounds, *energy threshold* and *size threshold*, and is discarded if it cannot result in a better solution than the best one found so far.

First, the algorithm is informally explained using a simple example. Consider a workload $n = 12$ executing on a given platform consisting of four heterogeneous processors ($p = 4$). Figure 4.2 shows the discrete dynamic energy functions, $E = \{e_0(x), \cdots, e_3(x)\}$, with a cardinality of 14 ($m = 14$), as inputs to *HEOPTA*. Figure 4.3 shows the discrete dynamic energy functions which are stored as arrays in non-decreasing order of energy consumption.

To solve the *HEOPT* problem and find the optimal workload distribution, a straightforward approach is to explore a full solution tree in order to build all combinations and finally select a workload distribution that its dynamic energy consumption is minimum. The tree explored by such a naive approach is shown in Figure 4.4 which contains all the combinations for $n = 12$ and $p = 4$. Due to the lack of space, the tree is shown partially.

The naive algorithm starts tree exploration from the root, which is the only node at the level L_0 of the tree. The root node is labelled by 12 which represents the whole workload to be distributed between 4 processors $\{P_0, P_1, P_2, P_3\}$. Then, fifteen ($= m + 1$) problem sizes, including a zero prob-

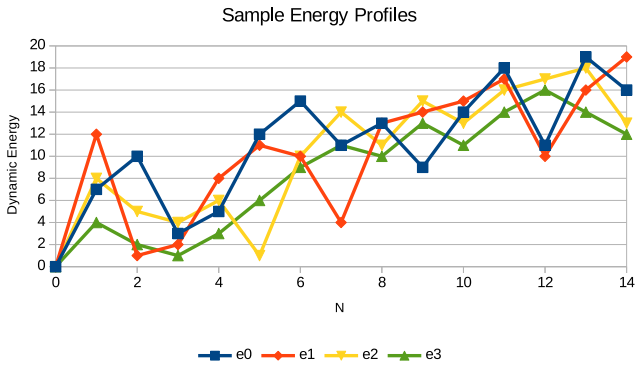

Figure 4.2: Dynamic energy functions of a sample application against problem size executing on an assumed parallel machine which consists of 4 processors.

	3	4	1	9	2	7	12	5	8	10	6	14	11	13
$e_0(x)$	3	5	7	9	10	11	11	12	13	14	15	16	18	19
	2	3	7	4	6	12	5	1	8	9	10	13	11	14
$e_1(x)$	1	2	4	8	10	10	11	12	13	14	15	16	17	19
	5	3	2	4	1	6	8	10	14	7	9	11	12	13
$e_2(x)$	1	4	5	6	8	10	11	13	13	14	15	16	17	18
	3	2	4	1	5	6	8	7	10	14	9	11	13	12
$e_3(x)$	1	2	3	4	6	9	10	11	11	12	13	14	14	16

Figure 4.3: Example: The sample dynamic energy functions, shown in Figure 4.2, which are stored in array data structures. Each array is sorted in non-decreasing order of dynamic energy consumption.

lem size along with all problem sizes in the dynamic energy function $e_0(x)$, are assigned to the processor P_0 one at a time. There is no assumption on the order of giving problem sizes to the processor, but we assign them in a non-decreasing order of their dynamic energy consumption. As shown in Figure 4.4, problem sizes $\{0, 3, 4, 1, 9, 2, 7, 12, 5, 8, 10, 6, 14, 11, 13\}$ are assigned to P_0 one after another at level L_0. Therefore, the root is expanded into 15 children. The value, which labels an internal node at level L_1 (root's children), determines the remaining workload to be distributed between pro-

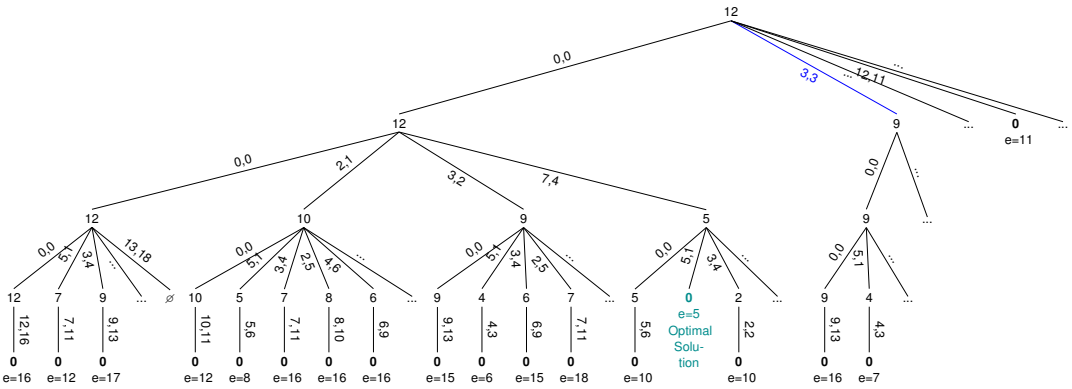

Figure 4.4: Applying naive approach to examine all combinations and select a workload distribution with the minimum dynamic energy consumption of parallel execution for a workload size of 12 *on* 4 *heterogeneous processors.*

cessors $\{P_1, P_2, P_3\}$.

Similarly, each child node of the root in the next level L_1 turns into a root of a sub-tree, which is a solution tree to solve *HEOPT* for the remaining workload between three processors $\{P_1, P_2, P_3\}$. Each edge, which connects the root and its child, is labelled by the problem size assigned to P_0 and its dynamic energy consumption. For example, the blue edge in Figure 4.4, which is labelled by $(3, 3)$, illustrates that a workload of size 3 is given to P_0 and it consumes 3 energy unit to execute this workload by the processor. The child node, connected by this edge, is labelled by 9, which is the remaining workload $(= 12 - 3)$ to be distributed between processors $\{P_1, P_2, P_3\}$.

In Figure 4.4, the leaf node at level L_1 labelled by 0 represents a solution leaf. Generally, any leaf node labelled by 0 illustrates one of the possible solutions, where its dynamic energy consumption is calculated as the summation of the consumed energies labelling the edges in the path connecting the root and the solution leaf. For example, the dynamic energy consumption of the solution represented by the *solution leaf* labelled by green 0, which is connected to the root by three edges $\{(0, 0), (7, 4), (5, 1)\}$, will be equal to $0 + 4 + 1 = 5$. It should be mentioned that P_0 and P_3 are given a zero problem size in this solution.

No-solution leaves are labelled by $\varnothing$. As an example, consider the *no-solution* leaf at level L_3. The path from this node to the root includes three

edges $\{(0,0),(0,0),(13,18)\}$. The corresponding workload distribution leads to no-solution because the sum of the workloads assigned to P_0, P_1 and P_2 will be equal to 13, which exceeds the total workload of 12.

In the same manner, all internal nodes in the tree are explored from the root to a leaf. The expansion of a node in the solution tree results in either 15 children (or $m+1$ in general case) or just one child where, In the latter case, the child is always a leaf. There are two types of leaves: *solution* leaves, labelled by 0, and *no-solution* leaves, labelled by $\varnothing$. Each internal node at level L_i, labelled by a positive number w, becomes a root of a solution tree for distribution of the workload w between processors $\{P_i, \cdots, P_{p-1}\}$ and is therefore explored recursively.

Finally, a distribution minimizing the dynamic energy consumption will be returned as the optimal solution. In this example, the workload distribution $\{(0,0),(7,4),(5,1),(0,0)\}$, represented by the green *Optimal Solution* leaf with the consumed dynamic energy of 5, is returned as the optimal solution.

Since the computational complexity of this naive algorithm is exponential, we propose *HEOPTA*, an efficient recursive sequential algorithm, with a polynomial complexity. The algorithm utilizes a number of optimizations to prevent the examination of all nodes and consequently does not explore all the paths in the tree.

HEOPTA starts the exploration process with sorting the discrete dynamic energy functions in non-decreasing order of energy consumption as shown in Figure 4.3. Next, it obtains the load-equal distribution for $n = 12$ as well as its dynamic energy consumption, stored in the variable ε, which is named *energy threshold*. The load-equal distribution allocates each processor the same workload of size $\frac{n}{p}$ (assuming n is divisible by p). *HEOPTA* will not examine data points with the dynamic energy consumption greater than or equal to the energy threshold. In the example, ε will be initialized by 10 ($\sum_{i=0}^{3} e_i(\frac{12}{4}) = (3+2+4+1) = 10$). Therefore, data points with dynamic energy consumption less than 10 will only be considered, forming the reduced search space. These data points are shown in grey cells in Figure 4.5. During the execution of *HEOPTA*, the energy threshold ε will be updated every time a solution with less dynamic energy consumption is found representing thus the

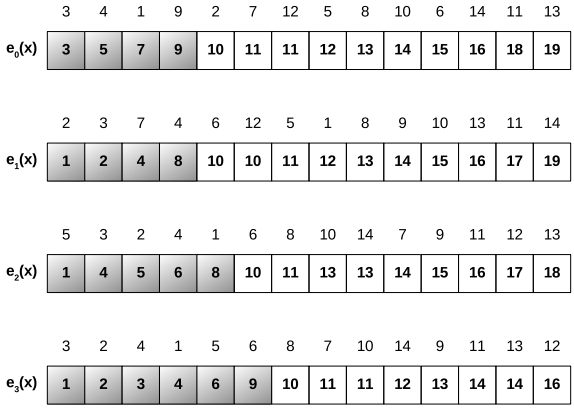

Figure 4.5: Example: Applying load-equal energy threshold and removing some data points from the search space.

consumed energy of the currently most energy saving solution.

To shrink the search space further, *HEOPTA* assigns each level of the tree a size threshold where the size threshold $\sigma_i, i \in \{0, \ldots, p-1\}$, represents the maximum workload can be executed in parallel on processors $\{P_i, \cdots, P_{p-1}\}$ so that the dynamic energy consumption by every processor $\{P_i, \cdots, P_{p-1}\}$ is less than ε. In this example, the maximum workloads with the dynamic energy consumptions less than $\varepsilon = 10$ in the dynamic energy functions for processors P_0, P_1, P_2, and P_3 are 9, 7, 5 and 6 respectively. The size threshold vector, σ contains four elements, $\sigma = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3\}$, where the size threshold for L_3 (σ_3) is equal to 6, σ_2 is 11 ($= \sigma_3 + 5$), σ_1 is set to 18 ($= \sigma_2 + 7$), and finally σ_0 would be 27 ($= \sigma_1 + 9$). Once ε changes, the size threshold array σ is also updated using the new ε.

HEOPTA then starts exploring the solution tree in the left-to-right depth-first order as shown in Figure 4.4. First, zero problem size is given to processors P_0 and P_1, where the remaining workload to be distributed between processors P_2 and P_3 is equal to 12. The node labelled by 12 in L_2 exceeds its corresponding size threshold σ_2, which is equal to 11. Henceforth, it cannot lead to any solution, which would be energy saver than the currently best (load-equal) solution with total dynamic energy consumption ε, and the node is not expanded. Therefore, the subtree highlighted in red in Figure 4.6 is cut and not explored. We call this key optimization operation *Cut*.

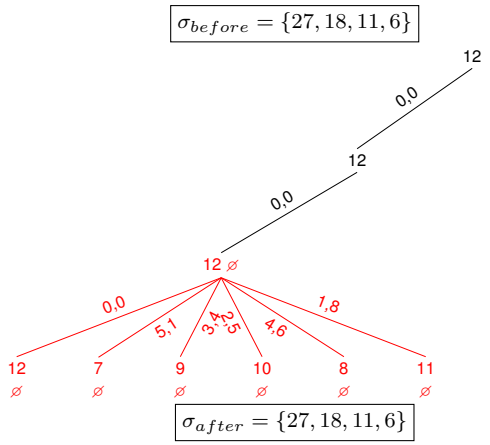

Figure 4.6: Example: Applying size threshold which results in cutting some branches, which do not give any solution, from the search tree.

To emphasize how σ changes during the *HEOPTA* execution, we show its value before and after each discussed step of the algorithm. As the *Cut* operation does not change ε, it also will not change σ, which is illustrated in Figure 4.6.

In its turn in the left-to-right order, as shown in Figure 4.7, next node to explore would be 10 at level L_2. The expansion of this node generates one solution (leaf in the tree labelled by 0). For the generated solution, the following operations will be performed:

- The energy threshold ε is updated.
- If ε decreases, the data points in the dynamic energy functions, whose dynamic energy is greater than or equal to the updated energy threshold, are removed from the search space, and the vector σ of size thresholds is updated.
- The solution is saved in the memory.

As an example, consider the solution $\{(0,0),(2,1),(5,1),(5,6)\}$ with an energy consumption of 8 (see Figure 4.7). The energy threshold, ε, is updated to 8. Based on the new energy threshold, the number of data points to be examined in the dynamic energy functions is reduced. This is illustrated in the

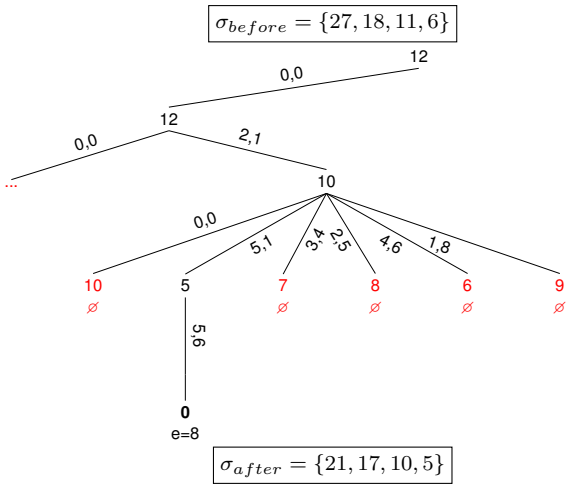

Figure 4.7: Example: Applying Cut *and* Save *optimizations.*

Figure 4.8, where fewer data points are required to be examined compared to Figure 4.5. The vector of size thresholds, σ, is updated to $\{21, 17, 10, 5\}$. The solution is memorized, which includes saving the information pertaining to all the levels except for the first and the last and the levels whose consumed energies go beyond ε. Thus, the information that is saved is level-specific. For L_1, the memorized information includes the problem size assigned to P_1, which is 2 and the total dynamic energy consumption of the solution for processors $\{P_1, P_2, P_3\}$, which is 8, The same is done for L_2. The memorized information includes the problem size assigned to P_2, which is 5 along with the total dynamic energy consumption of the solution for processors $\{P_2, P_3\}$, which is equal to 7. We call this key operation, *Save*. In the figure, red nodes have been cut using the size threshold.

HEOPTA, progressing in the left-to-right depth-first order, examines next node which is 9 at level L_2 as shown in Figure 4.9.

The algorithm finds a new solution with dynamic energy consumption of 6 including edges $\{(0, 0), (3, 2), (5, 1), (4, 3)\}$ (Figure 4.9). The energy threshold, ε, is updated to 6. The vector of size thresholds, σ, is updated to $\{20, 16, 9, 4\}$. For L_1, the memorized information includes the problem size assigned to P_1, which is 3 and the total dynamic energy consumption of the solution for processors $\{P_1, P_2, P_3\}$, which is 6. For L_2, the memorized in-

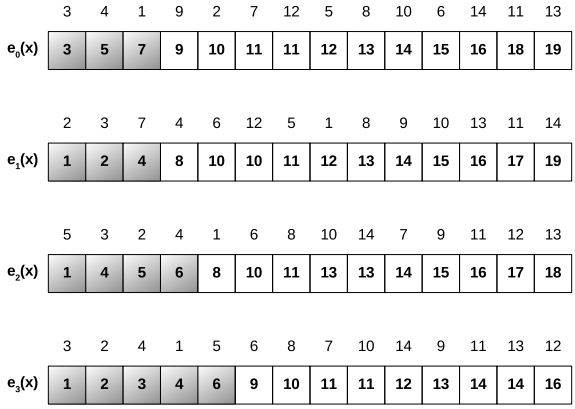

Figure 4.8: Example: Applying the updated energy threshold and removing more data points from the search space.

formation includes the problem size assigned to P_2, which is 5, and the total dynamic energy consumption of the solution for processors $\{P_2, P_3\}$, which is equal to 4. The nodes 9, 6 and 7 are cut by applying the size threshold.

After backtracking to the node 12 at level L_1, next node to be examined is the node 5 at level L_2 (Figure 4.10). The expansion of this node results in three solutions where the minimum one is $\{(0,0),(7,4),(5,1),(0,0)\}$ with a dynamic energy consumption of 5. The algorithm updates the energy threshold, ε, to 5. As the energy threshold decreased, the vector of size thresholds is updated to $\{19,16,9,4\}$. The solution is then stored in the memory. For L_1, the memorized information includes the problem size assigned to P_1, which is 7, and the total dynamic energy consumption of the solution for processors $\{P_1, P_2, P_3\}$, which is equal to 5. For L_2, the memorized information includes the problem size assigned to P_2, which is 5, and the total dynamic energy consumption of the solution for processors $\{P_2, P_3\}$, which is equal to 1.

The algorithm backtracks to L_1. However, there is no further data point in $e_1(x)$ to be examined (Updating ε has removed useless data points from the search space). Thus, it again backtracks to L_0. *HEOPTA* gives the problem size 3 to P_0 and 0 to P_1, coming to the node 9 at level L_2 as illustrated in Figure 4.11. Here, as the optimal distribution of the workload 9 between processors P_2 and P_3 has been already found and memorized, the best solution coming out of the node 9 at level L_2 will be just retrieved from the memory. We call this

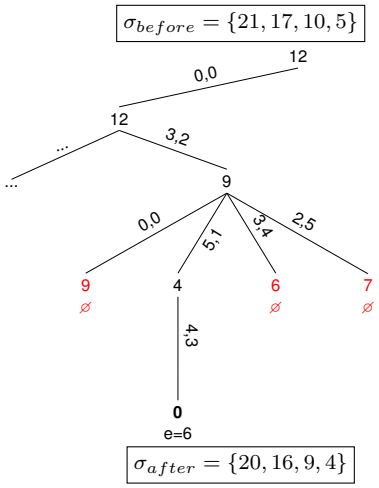

Figure 4.9: Example: Applying Cut *and* Save *optimizations.*

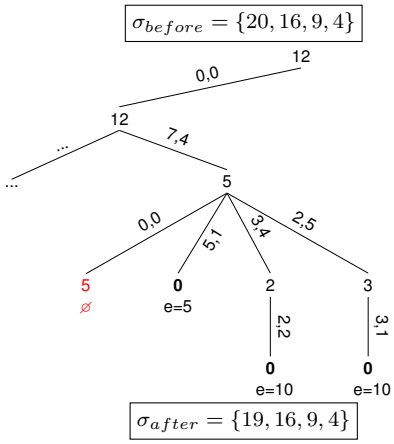

Figure 4.10: Example: Keeping on expanding the search tree using HEOPTA*.*

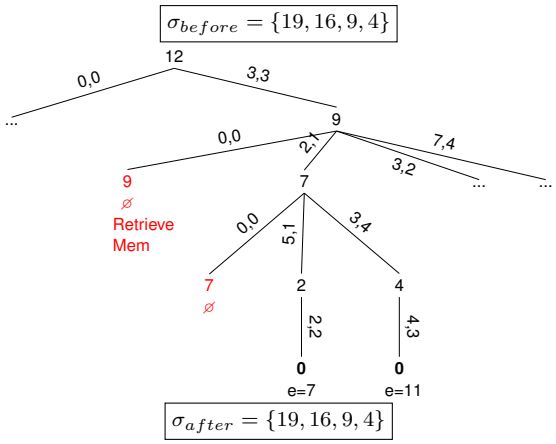

Figure 4.11: Example: Termination of HEOPTA.

key operation, READMEMORY. The dynamic energy consumption of retrieved solution using P_2 and P_3 equals 4. That is, the whole solution would have a consumed energy of 7 $(3 + 4)$ which is greater than current energy threshold ($\varepsilon = 5$). Thus, this solution is ignored. The algorithm will examine the problem sizes 2, 3 and 7 at level L_1 which lead to no solution with dynamic energy consumption less than 5 and then backs to the root. The reduction of the search space, by updating ε, results in the situation where no more data points in the dynamic energy functions are left for further exploration. Therefore, the algorithm terminates.

The optimal dynamic energy consumption is given by the last value of the energy threshold, which is 5 in this example. The optimal workload distribution is given by the workload distribution associated with this energy threshold, which is $\{(0, 0), (7, 4), (5, 1), (0, 0)\}$ in this example. So, *HEOPTA* found the optimal solution using the three key operations, which are a). *Cut*, b). *Save*, and c). READMEMORY.

In the next section, we give a pseudocode of our algorithm, which uses these key operations as the fundamental building blocks.

Like *HPOPTA*, a novel data-partitioning algorithm for performance optimization on modern heterogeneous platforms (Chapter 3), the proposed algorithm *HEOPTA* is based on the branch-and-bound solution technique. However, while *HPOPTA* solves *min-max* INLP single-objective optimization prob-

lems, *HEOPTA* deals with *min-sum* INLP ones. Therefore, the bounding criteria and the memorized information are different in these algorithms.

4.5 Formal Description of *HEOPTA*

Algorithm 3 shows the pseudocode of *HEOPTA*. It takes as inputs: the problem size, n, the number of heterogeneous processors, p, and an array of p discrete dynamic energy functions, $E = \{E_0, E_1, \cdots, E_{p-1}\}$. E_i represents the dynamic energy function of processor P_i and consists of m pairs (x_{ij}, e_{ij}), $j \in [0, m)$ where x_{ij} is the j-th problem size in the function, and e_{ij} represents the amount of dynamic energy consumed by P_i to run x_{ij}. *HEOPTA* returns two outputs: the optimal workload distribution, X_{opt}, and its optimal dynamic energy consumption, e_{opt}. It should be noted that the number of processors selected by the algorithm (processors with non-zero workloads) in the optimal workload distribution may be less than p.

The algorithm first sorts each profile in non-decreasing order of dynamic energy consumption (Line 2). After that, the array X_{opt} and the energy threshold ε are initialized to the load-equal distribution and its corresponding dynamic energy consumption, respectively (Lines 3-5). The vector of size thresholds, σ, is then determined using the function SIZETHRESHOLDCALC (Line 6).

In line 7, the data structure for saving solutions, matrix Mem, which consists of $(p-2) \times (n+1)$ elements, is initialized. It will save the solutions found for processors $\{P_1, \cdots, P_{p-2}\}$. Next, *HEOPTA* invokes the recursive routine HEOPTA_KERNEL to find the optimal workload distribution.

Function GETENG(E_i, x) (called in Line 5) returns the dynamic energy consumption of a problem size x running on P_i (The value is extracted from E_i). It returns 0 when x equals 0. It should be mentioned that pseudocodes of all functions, used in Algorithms 3 and 4, and the structure of Mem can be found in Appendix C.

HEOPTA is a one-dimensional data-partitioning algorithm. Nevertheless, as explained in Chapter 3, it can be directly employed to 2D or 3D problems in case the dimensionality can be reduced to 1D.

Algorithm 3 Algorithm Finding Optimal Workload Distribution of Size n for Minimizing Dynamic Energy Consumption

1: **function** HEOPTA($n, p, E, X_{opt}, e_{opt}$)
INPUT:
Problem size, $n \in \mathbb{Z}_{>0}$
Number of processors, $p \in \mathbb{Z}_{>0}$
Dynamic energy functions, $E = \{E_0, ..., E_{p-1}\}$,
$E_i = \{(x_{ij}, e_{ij}) \mid i \in [0, p), j \in [0, m), x_{ij} \in \mathbb{Z}_{>0}, e_{ij} \in \mathbb{R}_{>0}\}$.
OUTPUT:
Optimal workload distribution, $X_{opt} = \{x_{opt}[0], ..., x_{opt}[p-1]\}$,
$x_{opt}[i] \in \{\bigcup_{j=0}^{m-1} x_{ij} \cup \{0\}\}, i \in [0, p)$.
Total dynamic energy consumption, $e_{opt} \in \mathbb{R}_{>0}$

2: $E \leftarrow E \cup Sort_{\uparrow}(E)$
3: $x_{opt}[i] \leftarrow \frac{n}{p}, \forall i \in [0, p-1]$
4: $x_{opt}[i] \leftarrow x_{opt}[i] + 1, \forall i \in [0, n\%p)$
5: $\varepsilon \leftarrow \sum_{i=0}^{p-1}$ GETENG($E_i, x_{opt}[i]$)
6: $\sigma \leftarrow$ SIZETHRESHOLDCALC(p, E, ε)
7: $Mem[i][j] \leftarrow \emptyset, \forall i \in [1, \cdots, p-2], j \in [0, \cdots, n]$
8: HEOPTA_KERNEL($n, p, 0, E, \varepsilon, \sigma, 0, X_{cur}, Mem, X_{opt}$)
9: $e_{opt} \leftarrow \varepsilon$
10: **return** (X_{opt}, e_{opt})
11: **end function**

The correctness proof of *HEOPTA* is presented in Appendix C. We also prove that the computational complexity of *HEOPTA* is $O(m^3 \times p^3)$, and its memory complexity is $O((m+n) \times p)$. Because of only considering the operation *Save* to obtain the complexity of *HEOPTA*, its practical computational cost is considerably less than the theoretical one, $O(m^3 \times p^3)$.

4.5.1 Recursive Algorithm $HEOPTA\_Kernel$

HEOPTA_KERNEL (Algorithm 4) is a recursive function, deploying the key three operations, *Cut*, *Save* and READMEMORY to solve *HEOPT* problem efficiently. The variable c indicates the level of a node which is being processed in a solution tree. It is initialized to 0 in the first invocation of HEOPTA_KERNEL, and the next recursive invocation deals with L_1 (i.e. $c = 1$) and so on. The vector $X_{opt} = \{x_{opt}[0], \cdots, x_{opt}[p-1]\}$ holds the best solution found so far where its dynamic energy consumption is in ε. The array X_{cur} is used to hold problem sizes currently assigned to processors $P_i (i \in [0, p-1])$.

The function CUT(n, σ_c), applying the key operation *Cut*, compares the workload n with the corresponding size threshold σ_c to decide whether to expand the node or cut the subtree in level c (Lines 2-4).

Lines 5-11 process the solutions found in the last level L_{p-1}. Generally, once a solution is found, the routine PROCESSSOLUTION is invoked to perform the following operations :

- If X_{cur} is energy saver than the current best solution, X_{opt}, the energy threshold ε will be reduced to the dynamic energy consumption of X_{cur}, and X_{opt} will be updated to X_{cur}.
- Should ε decreases, the size threshold vector σ is correspondingly updated.
- The operation *Save* is invoked to save X_{cur} in the memory.

Prior to expanding a node with a label of n at a given level c, the function READMEMORY is called to retrieve the solution for n on processors $\{P_c, \cdots, P_{p-1}\}$, provided it exists (Lines 12-20).

The optimal and intermediate solutions are stored in Mem. A memory cell which contains the optimal distribution is labelled *Finalized*. The variable *status* determines the type of the retrieved solution. If there is no solution stored in a finalized cell or the total amount of dynamic energy consumption for the retrieved solution is greater than or equal to ε (given by the status, *NOT_SOLUTION*), we return from HEOPTA_KERNEL. If the stored solution in the Mem is the optimal one (given by the status, *SOLUTION*), the retrieved solution is used and the process returns from HEOPTA_KERNEL. If none of the above cases takes place, it means that the node has not already been examined, and the routine starts expanding the current node by scanning the dynamic energy profile E_c from left to right.

The variable idx, ranging from -1 to $m-1$, determines indexes of data points in the sorted dynamic energy functions. Line 21 initializes idx to -1 and $x_{c\ idx}$ to zero. Generally, $x_{c\ idx}$ determines the idx-th problem size in profile E_c, in case idx is not -1.

The $while$ loop (Lines 22-35) scans the dynamic energy profile E_c from left to right examining data points with dynamic energy consumption less than the energy threshold ε. The array $X_{cur} = \{x_{cur}[0], \cdots, x_{cur}[p-1]\}$ where $x_{cur}[i] \in \{\bigcup_{j=0}^{m-1} x_{ij} \cup \{0\}\}$, $i \in [0, p)$, is used to store problem sizes currently

assigned to processors P_i. In each iteration, the data point idx is extracted from E_c, and its workload, which is $x_{c\ idx}$, is stored in array X_{cur} (Line 23). If this workload ($x_{c\ idx}$) is equal to n, we found a solution. In this case, the solution is processed using PROCESSSOLUTION, otherwise, if $x_{c\ idx}$ is less than n, HEOPTA_KERNEL is re-invoked to solve *HEOPT* for the remaining workload $n - x_{c\ idx}$ at the next level L_{c+1} (Lines 24-30). If $x_{c\ idx}$ greater than n, this data point is declined and the next one is processed.

Lines 31-34 check if the algorithm reaches the end of the function E_c. If this is the case, the *while* loop (Line 22-35) terminates, and the corresponding memory cell is finalized (Line 36). Otherwise, idx is incremented moving to the next data point in the dynamic energy function E_c.

Algorithm 4 Algorithm of Recursive Kernel Invoked by Algorithm 3

1: **function** HEOPTA_KERNEL($n, p, c, E, \varepsilon, \sigma, X_{cur}, Mem, X_{opt}$)
2: **if** CUT(n, σ_c) **then**
3: **return**
4: **end if**
5: **if** $c = p - 1$ **then**
6: **if** GETENG(E_c, n) $< \varepsilon$ **then**
7: $x_{cur}[p-1] \leftarrow n$
8: PROCESSSOLUTION($p, E, \varepsilon, \sigma, X_{cur}, Mem, -1, X_{opt}$)
9: **end if**
10: **return**
11: **end if**
12: **if** $c > 0 \wedge c \leq p - 2$ **then**
13: $status \leftarrow$ READMEMORY($n, p, c, \varepsilon, E, X_{cur}, Mem, idx$)
14: **if** $status = NOT\_SOLUTION$ **then**
15: **return**
16: **else if** $status = SOLUTION$ **then**
17: PROCESSSOLUTION($p, E, \varepsilon, \sigma, X_{cur}, Mem, c, X_{opt}$)
18: **return**
19: **end if**
20: **end if**
21: $idx \leftarrow -1, x_{c\ idx} \leftarrow 0$
22: **while** GETENG($E_c, x_{c\ idx}$) $< \varepsilon$ **do**
23: $x_{cur}[c] \leftarrow x_{c\ idx}$
24: **if** $x_{c\ idx} = n$ **then**
25: $x_{cur}[i] \leftarrow 0, \forall i \in [c+1, \cdots, p-1]$
26: PROCESSSOLUTION($p, E, \varepsilon, \sigma, X_{cur}, Mem, -1, X_{opt}$)
27: **end if**
28: **if** $n > x_{c\ idx}$ **then**
29: HEOPTA_KERNEL($n - x_{c\ idx}, p, c+1, E, \varepsilon, \sigma, X_{cur}, Mem, X_{opt}$)
30: **end if**
31: **if** $idx + 1 = m$ **then**
32: **break**
33: **end if**
34: $idx \leftarrow idx + 1$
35: **end while**
36: MAKEFINAL($Mem[c][n]$)
37: **end function**

Table 4.1: HCLServer01: Specifications of the Intel Haswell multicore CPU, Nvidia K40c, and Intel Xeon Phi 3120P.

Intel Haswell E5-2670V3	
No. of cores per socket	12
Socket(s)	2
CPU MHz	1200.402
L1d cache, L1i cache	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30720 KB
Total main memory	64 GB DDR4
Memory bandwidth	68 GB/sec
NVIDIA K40c	
No. of processor cores	2880
Total board memory	12 GB GDDR5
L2 cache size	1536 KB
Memory bandwidth	288 GB/sec
Intel Xeon Phi 3120P	
No. of processor cores	57
Total main memory	6 GB GDDR5
Memory bandwidth	240 GB/sec

4.6 Experimental Results of *HEOPTA*

In this section, we examine the accuracy of the proposed *additivity* approach for determining dynamic energy functions and present the experimental results of the proposed algorithm for dynamic energy optimization, *HEOPTA*, using two well-known data-parallel parallel applications, Matrix Multiplication and 2D FFT.

4.6.1 Experimental Platform and Applications

We conduct all experiments on a cluster consisting of two nodes. The first heterogeneous node, *HCLServer01*, consists of an Intel Haswell multicore CPU which is integrated with an Nvidia K40c GPU and an Intel Xeon Phi 3120P. The specifications of this platform are given in Table 4.1. Another hybrid server, *HCLServer02*, includes an Intel Skylake multicore CPU hosting one Nvidia P100 PCIe GPU, and its specifications can be found in Table 4.2.

Table 4.2: HCLServer02: Specifications of the Intel Skylake multicore CPU and Nvidia P100 PCIe.

Intel Xeon Gold 6152	
Socket(s)	1
Cores per socket	22
L1d cache, L1i cache	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30976 KB
Main memory	96 GB
NVIDIA P100 PCIe	
No. of processor cores	3584
Total board memory	12 GB CoWoS HBM2
Memory bandwidth	549 GB/sec

We group the processing units of both the platforms into five loosely-coupled abstract processors, as explained in Section 4.2.1. We name them as *CPU_1*, *GPU_1*, *Phi_1* abstract processors, on *HCLServer01*, and *CPU_2* and *GPU_2* abstract processors, on *HCLServer02*.

The accuracy of our additivity approach and the efficiency of *HEOPTA* are elucidated using two widely known data-parallel applications, Matrix Multiplication and 2D discrete Fourier Transform (2D FFT). The Matrix Multiplication application (DGEMM) computes $C = \alpha \times A \times B + \beta \times C$, where A, B, and C are respectively dense matrices of size $m \times n$, $n \times n$, and $m \times n$ and α and β are constant floating-point numbers. The application 2D FFT computes the Fourier transform of a complex matrix of size $m \times n$. These applications are configured to run on the heterogeneous platforms *HCLServer01* and *HCLServer02*. Each application consists of three different kernels, one for CPU, one for GPU, and one for Xeon Phi abstract processors.

For CPUs, the Matrix Multiplication application uses DGEMM routine provided in Intel MKL BLAS [143]. For GPUs and the Intel Xeon Phi, the application employs two packages, *ZZGemmOOC* [144] and *XeonPhiOOC* [146], that perform out-of-card matrix multiplication of large dense matrices on them. The *ZZGemmOOC* out-of-card package reuses CUBLAS [145] for in-card DGEMM calls, and *XeonPhiOOC* out-of-card package reuses MKL BLAS [143] for in-card DGEMM calls. We will explain the structure of these out-of-card pack-

ages in Chapter 6. The Intel MKL and CUDA versions used on *HCLServer01* are respectively 2017.0.2 and 7.5. The CUDA version 9.2.148 is installed on *HCLServer02*.

The 2D FFT application invokes Intel MKL FFT [147] for multicore CPUs and Xeon Phi, and CUFFT [148] is used for the Nvidia GPUs. All computations for the application are in-card.

For measuring dynamic energy consumption, each node (*HCLServer01* and *HCLServer02*) is facilitated with one WattsUp Pro power meter which sits between the wall A/C outlets and the input power sockets of the node. Each power meter captures the total power consumption of one node. We use *HCLWattsUp* API [156], which gathers the readings from the power meters to determine the average power and energy consumption during the execution of an application for the whole node. *HCLWATTSUP* has no extra overhead and therefore does not influence the energy consumption of the application kernel.

4.6.2 Experimental Analysis

In this section, we experimentally validate the accuracy of the additivity approach. The additivity methodology to build the discrete dynamic energy function of each abstract processor has been explained in Section 4.2.1. To verify if *additivity* hypothesis is valid, we build four profiles for *HCLServer01* (one parallel and one for each of the three abstract processors), and three profiles for *HCLServer02* (one parallel and one for each of the two abstract processors). Our approach on how to instrument computational kernels in a hybrid application and separately measure their execution times and dynamic energies will be explained in detail in Appendix A.

Then, the proposed algorithm, *HEOPTA*, is examined by using two sets of experiments. For the first set, we compare the dynamic energy consumption of solutions determined by *HEOPTA* over load-balanced workload distributions. Load-balanced solutions are referred to as workload distributions with almost the equal execution time for each abstract processor. The number of active processors in load-balanced distributions may be less than the total number of processors. The percentage energy saving against load-balancing algo-

rithm is obtained as follows: $Energy\_Saving_{balance}(\%) = \frac{e_{balance}-e_{heopta}}{e_{heopta}} \times 100$, where $e_{balance}$ and e_{heopta} are respectively the dynamic energy consumptions of distributions determined by load-balancing and *HEOPTA* algorithms.

For the second set of experiments, we examine the interplay between dynamic energy optimization and performance optimization using *HPOPTA* workload distribution. As explained in Chapter 3, *HPOPTA* is a novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous HPC platforms. We build performance profiles of each abstract processor for the Matrix Multiplication and 2D FFT applications, as explained in Chapter 3, and then use *HPOPTA* to determine workload distributions minimising performance. The percentage energy saving of *HEOPTA* solutions over *HPOPTA* ones is obtained as follows: $Energy\_Saving_{hpopta}(\%) = \frac{e_{hpopta}-e_{heopta}}{e_{heopta}} \times 100$, where e_{hpopta} represents the dynamic energy consumptions of distributions determined by *HPOPTA* algorithm.

Analysing using Matrix Multiplication

Figures 4.12 and 4.13 represent discrete dynamic energy and performance functions of the Matrix Multiplication application for *CPU_1*, *GPU_1*, and *Phi_1*, abstract processors in *HCLServer01*, and *CPU_2* and *GPU_2* abstract processors of *HCLServer02*. We need the speed functions to obtain e_{hpopta}. In these functions the problem sizes range from 64×10112 to 28800×10112 with a step size of 64 for the first dimension m. It should be mentioned that *HEOPTA* is capable to deal with any step size in discrete dynamic energy functions. For each data point in the functions, the experiments are repeated until sample means of all the five kernels running on the abstract processors fall in the confidence interval of 95%, and a precision of 0.1 (10%) is achieved. As shown in Figure 4.12, there is a marked drop in dynamic energy consumption once out-of-card computation on the abstract processor *Phi_1* starts.

Now, we examine the validity of the proposed additivity approach. Figure 4.14 shows the dynamic energy functions for parallel and combined executions of DGEMM running on all aforementioned abstract processors. Here, combined refers to the sum of dynamic energy consumption of all abstract

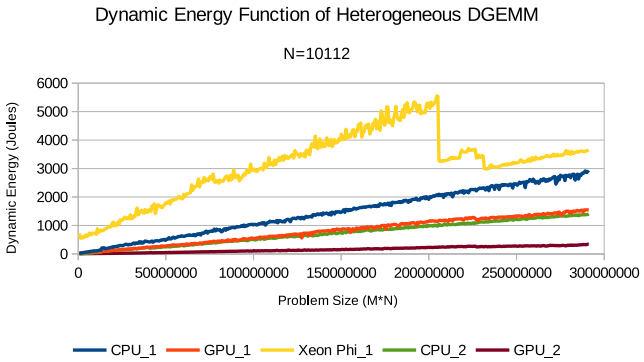

Figure 4.12: Dynamic energy functions of the heterogeneous Matrix Multiplication application executing on HCLServer01 *and* HCLServer02*.*

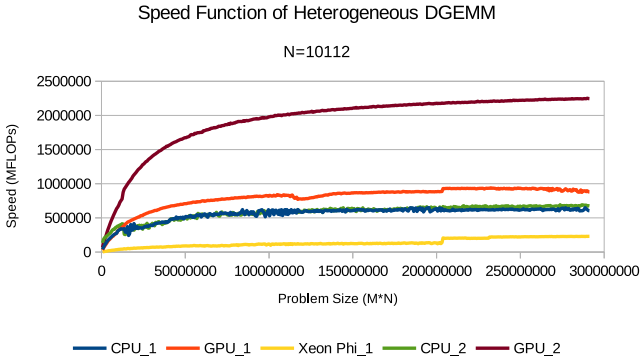

Figure 4.13: Speed functions of the heterogeneous Matrix Multiplication application executing on HCLServer01 *and* HCLServer02*.*

processors when running a given workload separately (i.e. one abstract processor is performing computations).

Table 4.3 summarizes the percentage difference of parallel to combined for the heterogeneous Matrix Multiplication application.

To elucidate the energy saving percentage given by *HEOPTA*, we create an experimental data set $\{64 \times 10112, 128 \times 10112, \cdots, 57600 \times 10112\}$. Figure 4.15 presents the dynamic energy consumption of the heterogeneous Matrix Multiplication when is executed using *HEOPTA* in comparison with load-balanced workload distribution. Minimum, average and maximum reduction in the dynamic energy consumption of *HEOPTA* over load-balancing algorithm, $Energy\_Saving_{balance}$, are 0%, 130%, and 257%, respectively. Zero percent-

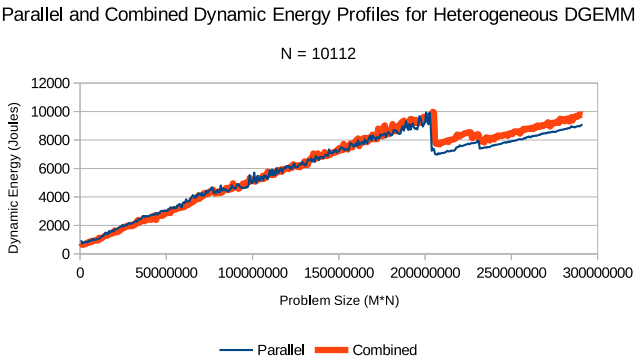

Figure 4.14: Parallel and Combined dynamic energy functions for the heterogeneous Matrix Multiplication application on HCLServer01 *and* HCLServer02.

Table 4.3: Percentage difference of dynamic energy consumption of parallel to combined for the heterogeneous Matrix Multiplication.

Platform	**Application**	**Min**	**Max**	**Average**
HCLServer01	DGEMM	0.026%	29.2%	6.38%
HCLServer02	DGEMM	0.012%	29.03%	3.8%
BOTH	DGEMM	0.004%	26.1%	5.9%

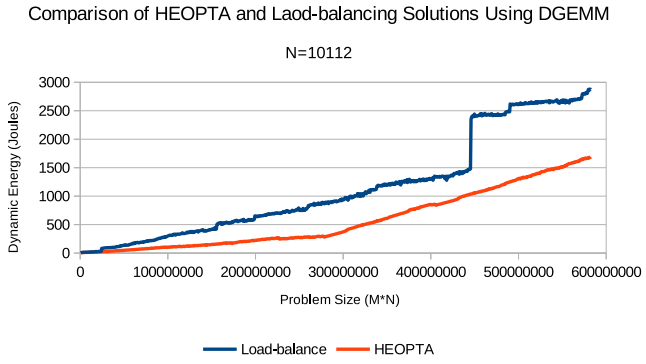

Figure 4.15: Dynamic energy consumption of the heterogeneous Matrix Multiplication application executed using HEOPTA *in comparison with load-balanced workload distribution on* HCLServer01 *and* HCLServer02.

age improvement represents that the same workload distribution is determined by *HEOPTA* and load-balancing algorithm.

Figure 4.16 compares *HEOPTA* over the dynamic energy consumption of workload distributions determined by *HPOPTA* using the Matrix Multiplication

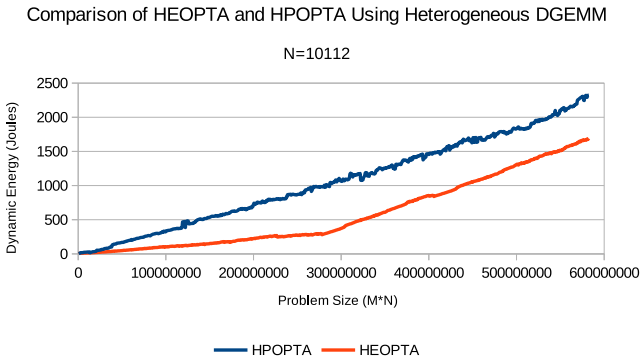

Figure 4.16: Dynamic energy consumption of the heterogeneous Matrix Multiplication executed using HEOPTA *in comparison with* HPOPTA *workload distribution on* HCLServer01 *and* HCLServer02*.*

application. Minimum, average and maximum for $Energy\_Saving_{HPOPTA}$ are respectively 0%, 145%, and 314%. According to these results, performance optimization increases dynamic energy consumption by an average of 145%.

Analysing using 2D FFT

In this section, we analyse the solutions returned by *HEOPTA* using our 2D FFT application. Figures 4.17 and 4.19 present the dynamic energy and performance functions of the five abstract processors. The abstract processor Phi_1 consumes ten times more dynamic energy than the other processors. To highlight complex shapes of dynamic energy functions, Figure 4.18 shows the dynamic energy functions of the 2D FFT application excluding Phi_1. In these functions, the problem sizes range from 1024×51200 to 10000×51200 with a step size of 16 for the first dimension m. It does not include problem sizes, which cannot be factored into primes less than or equal to 127. For these problem sizes, CUFFT gives failures for GPU. We follow the same methodology as the matrix multiplication application to collect each data point in these functions.

For validating the additivity approach, Figure 4.20 shows the parallel and combined dynamic energy functions of the 2D FFT application, and Table 4.4 represents the statistics for percentage difference of parallel to combined.

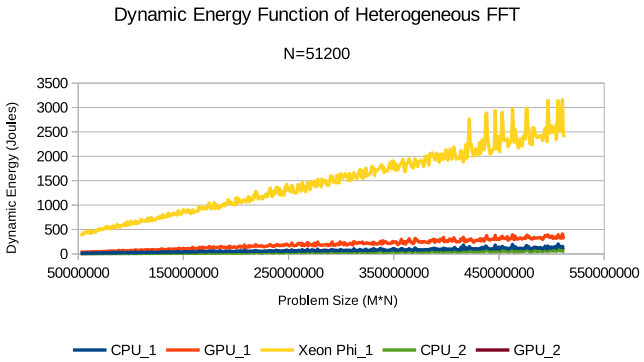

Figure 4.17: Dynamic energy functions of the heterogeneous 2D FFT application executing on HCLServer01 *and* HCLServer02*.*

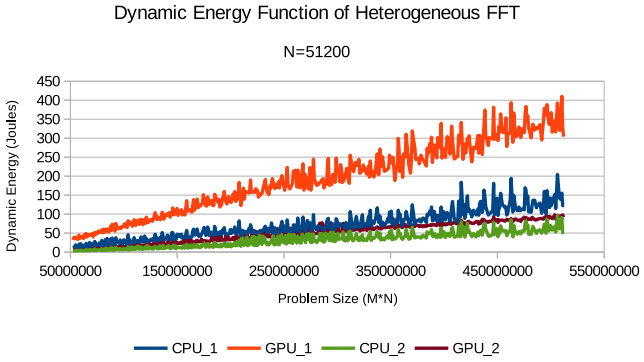

Figure 4.18: Dynamic energy functions of the heterogeneous 2D FFT application executing on HCLServer01 *and* HCLServer02*. In this figure, the dynamic energy profile for Phi_1 is ignored.*

The experimental data set to conduct our experiments with 2D FFT include data points $\{1024 \times 51200, 1040 \times 51200, \cdots, 20000 \times 51200\}$. Minimum, average and maximum dynamic energy reduction of *HEOPTA* over load-balancing algorithm, $Energy\_Saving_{balance}$, are 0%, 44%, and 105%, respectively. Figure 4.21 compares *HEOPTA* against the dynamic energy consumption of load-balanced workload distributions for the 2D FFT application. The minimum, average and maximum of $Energy\_Saving_{HPOPTA}$ are respectively 0%, 32%, 77%. Figure 4.22 shows *HEOPTA* dynamic energy consumption over *HPOPTA*. Regarding these results, one can conclude that performance optimization or 2D FFT increases dynamic energy consumption by an average

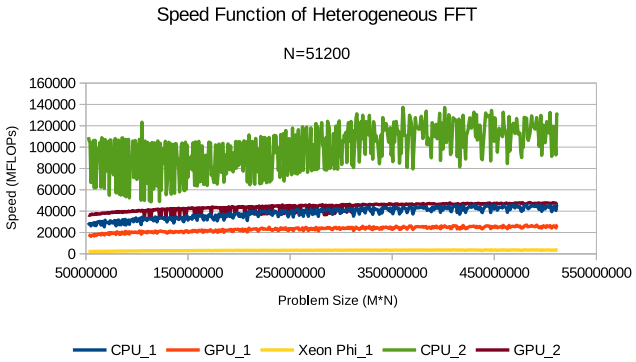

Figure 4.19: Speed functions of the heterogeneous 2D FFT application executing on HCLServer01 *and* HCLServer02*.*

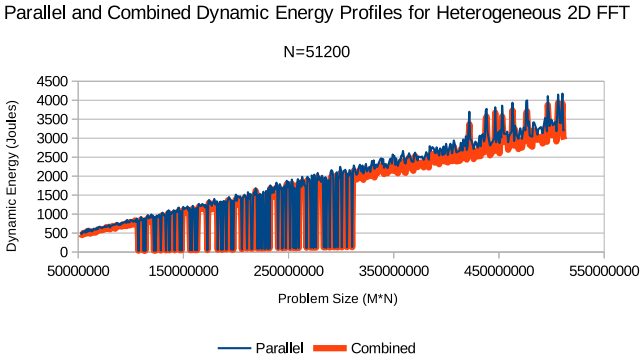

Figure 4.20: Parallel and Combined dynamic energy functions for the heterogeneous 2D FFT application on HCLServer01 *and* HCLServer02*.*

of 32%.

4.6.3 Observations

We find an average difference of 5.9% and 8.3% between parallel and combined dynamic energy functions on both *HCLServer01* and *HCLServer02* for DGEMM and FFT, respectively. One can observe that despite the percentage error, both parallel and combined profiles follow the same pattern for both applications.

According to the experimental results, the proposed algorithm demonstrates considerable improvements in average and maximum dynamic energy

Table 4.4: Percentage difference of dynamic energy consumption of parallel to combined for the heterogeneous 2D FFT application.

Platform	Application	Min	Max	Average
HCLServer01	2D-FFT	1.8%	18.4%	9.1%
HCLServer02	2D-FFT	0.02%	28.8%	12.4%
BOTH	2D-FFT	0.16%	24.7%	8.3%

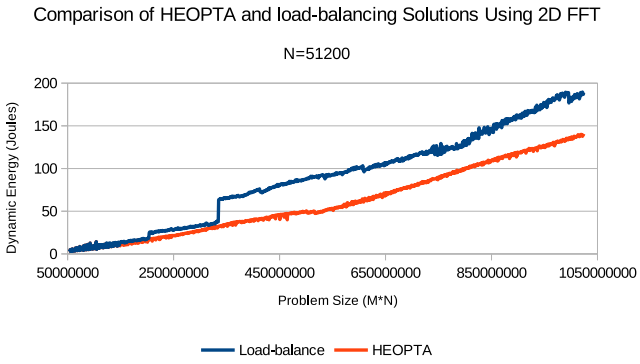

Figure 4.21: Dynamic energy consumption of the heterogeneous 2D FFT application executed using HEOPTA *in comparison with load-balanced workload distribution on* HCLServer01 *and* HCLServer02*.*

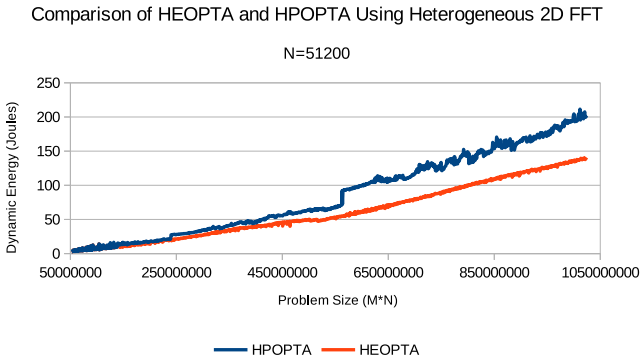

Figure 4.22: Dynamic energy consumption of the heterogeneous 2D FFT application executed using HEOPTA *in comparison with* HPOPTA *workload distribution on* HCLServer01 *and* HCLServer02*.*

consumptions for the two applications in comparison with the load-balancing and *HPOPTA* algorithms. In addition, in comparison with *HEOPTA*, performance optimization using *HPOPTA* negatively influences dynamic energy con-

sumption for both applications.

4.7 Summary

In this chapter, we explained that increasing the number of cores in a single die and also tight integrating of multi-core CPUs with many-core accelerators introduce new challenges to modelling and the optimization of data-parallel applications on these platforms for dynamic energy. We experimentally showed that there is a complex correlation between dynamic energy consumption and workload size where workload distribution has now become an important decision variable that should be taken into account in solving the energy optimization problem.

The first fundamental challenge to dynamic energy optimization is to accurately attribute the energy consumption of every single computational kernel in hybrid scientific applications executing on heterogeneous HPC platforms. To solve this problem, we proposed a novel solution methodology which is purely based on physical measurements and does not rely on any performance monitoring counter to build the dynamic energy profiles of computational kernels (abstract processors) separately. This methodology, which is in the early stage of development, facilitates modelling the dynamic energy consumption of data-parallel applications in terms of some discrete dynamic energy profiles which are functions of problem size.

We then formulated the problem of finding optimal distribution on heterogeneous HPC platforms and proposed a novel model-based data partitioning algorithm to minimize the dynamic energy consumption for the most general shapes of dynamic energy functions for parallel applications executing on heterogeneous HPC systems. The algorithm takes as input the workload size, n, a set of p discrete dynamic energy functions, which represent the dynamic energy consumption of p processors existing in the heterogeneous platform. The time complexity of the proposed algorithm is $O(m^3 \times p^3)$ where m and p respectively represent the maximum cardinality of input energy functions and the number of heterogeneous processors.

We studied the accuracy of the methodology and the optimality of solutions found by the proposed algorithm using two well-known data-parallel applications, matrix multiplication and 2D FFT. Regarding the experimental results, there is a great opportunity for application-level dynamic energy optimization on modern heterogeneous platforms when *workload distribution* is only decision variable. We also observed that optimizing for execution time led to a considerable increase in dynamic energy consumption. Thus, a bi-objective optimization algorithm can facilitate making a trade-off between the performance and the dynamic energy consumption of data-parallel applications executing on modern heterogeneous platforms.

The software implementation for *HEOPTA* is available at [157].

Chapter 5

Bi-objective Optimization of Data-parallel Applications on Heterogeneous HPC Platforms for Performance and Energy Using Workload Partitioning

Performance and energy are the two most dominant objectives for optimization on modern parallel platforms such as supercomputers and cloud computing infrastructures.

Existing algorithms solving bi-objective optimization problems for performance and energy on heterogeneous HPC platforms can be broadly classified into *system-level* [61, 62, 63, 10, 64, 47] and *application-level* [65, 66, 67, 49, 68, 69, 70, 71, 72, 73, 50, 51] categories. The dominant decision variable in the first category is DVFS, and application-level methods use application-level parameters such as the number of threads, the number of processors and task mapping (scheduling) as key decision variables. These methods take into account workload size as an application parameter but assume a linear relationship between performance and workload size and between energy consumption and workload size, and therefore, do not consider workload

distribution as a decision variable.

As highlighted in Chapters 1, 3 and 4, due to new complexities such as severe contention on shared resources and NUMA, workload distribution has now turned into an important decision variable that should not be ignored in solving performance and also energy optimization problems on modern heterogeneous HPC platforms. Furthermore, in Chapter 4, we experientially demonstrated that performance optimization negatively impacts the dynamic energy consumption of applications and vice versa. Therefore, it implies that there are markedly trade-off solutions for performance and dynamic energy when workload distribution is used as the decision variable.

In this chapter, we propose a novel model-based data partitioning algorithm, *HEPOPTA*, solving bi-objective optimization problem for execution time and dynamic energy (BOPPE), which has only one decision variable, *workload distribution*. The inputs to *HEPOPTA* are the problem size, n, the number of available heterogeneous processors, p, p discrete performance functions (one for each processor), and p discrete dynamic energy functions (one for each processor). It returns the globally Pareto-optimal solutions for performance and dynamic energy with a time complexity of $O(m^3 \times p^3 \times \log_2(m \times p))$, where m represents the maximum cardinality of the discrete functions. Each solution in the set is a distribution of workload n between the p heterogeneous processors. These sets of solutions contain the load-balanced solution in very few cases, and the rest of the solutions are load-imbalanced. To the best of our knowledge, none of the traditional approaches to optimization for performance and energy consider non-balanced solutions as optimal. In Appendix D, we will elucidate that the practical time complexity of *HEPOPTA* is enormously less than the proposed theoretical one.

We also present another algorithm, named *HTPOPTA*, which solves bi-objective optimization problem for performance and total energy. The inputs to this algorithm are the same as those for *HEPOPTA* and the base power of the platform. *HTPOPTA* reuses *HEPOPTA* to determine the globally Pareto-optimal solutions for performance and total energy, and its time complexity is the same as *HEPOPTA*. In this chapter, We show that the workload distribution that minimises the dynamic energy consumption will not necessarily minimise

the total energy consumption.

We experimentally analyse these algorithms using two data parallel applications, matrix multiplication and 2D fast Fourier transform on an HPC cluster of two heterogeneous nodes. We will show that the proposed data partitioning algorithms determine a better Pareto-optimal front containing all the load-imbalanced solutions that are totally ignored by load-balancing approaches. Therefore, unlike approaches looking for load-balanced solutions, solutions returned by the algorithms are, generally speaking, non-balanced.

5.1 Formulation of Heterogeneous Dynamic Energy-Performance Optimization Problem (HEPOPT)

Given a problem size n running on p heterogeneous processors so that their dynamic energy and performance functions are represented by $E = \{e_0(x), ..., e_{p-1}(x)\}$ and $T = \{t_0(x), ..., t_{p-1}(x)\}$ where $e_i(x)$ ($t_i(x)$), $i \in \{0, 1, \cdots, p-1\}$, is a discrete dynamic energy (performance) function with a maximum cardinality of m for processor P_i. The function $e_i(x)$ represents the amount of dynamic energy consumed by P_i to execute the problem size x, and $t_i(x)$ is the execution time of the problem size on this processor. Without loss of generality, we assume $x \in \{1, 2, \cdots, m\}$.

The bi-objective optimization problem is to find a workload distribution minimizing the execution time and dynamic energy consumption of computations during the parallel execution of the workload n using the p processors. The problem is formulated as follows:

$$HEPOPT(n,p,m,T,E): \min_{X} \quad \{\max_{i=0}^{p-1} t_i(x_i), \sum_{i=0}^{p-1} e_i(x_i)\}$$

Subject to:

$$\begin{aligned} &\sum_{i=0}^{p-1} x_i = n \\ &0 \leq x_i \leq m, \qquad i = 0, \cdots, p-1 \\ &\text{where} \quad p, n, m \in \mathbb{Z}_{>0}, \quad x_i \in \mathbb{Z}_{\geq 0}, \quad t_i(x), e_i(x) \in \mathbb{R}_{\geq 0} \end{aligned} \tag{5.1}$$

For each given workload distribution $X = \{x_0, \cdots, x_{p-1}\}$, *HEPOPT* calculates the parallel execution time, which is the time taken by the longest running processor to execute its workload, and the total dynamic energy consumption, which is equal to the summation of dynamic energies consumed by the p processors.

HEPOPT returns a set of Pareto-optimal solutions which determine the workload distributions. Each solution in the set is a distribution of workload n between the p heterogeneous processors, which, generally speaking, is not balanced. One or more processors in an optimal solution can be allocated a workload of size zero.

5.2 *HEPOPTA*: Algorithm Finding Globally Pareto-optimal Solutions for Dynamic Energy and Performance

In this section, we use a simple example to describe our proposed branch-and-bound algorithm, *HEPOPTA* (**H**eterogeneous **E**nergy-**P**erformance **OPT**imization **A**lgorithm), solving the problem *HEPOPT*. To shrink the search space, we define two bounding criteria, *energy threshold* and *size threshold*, explained later in this section.

Suppose there are four heterogeneous processors ($p = 4$) executing a

	2	1	3	4
$e_0(x)$	1	3	5	6
$t_0(x)$	2	5	6	5

	2	1	3	4
$e_1(x)$	1	2	6	9
$t_1(x)$	6	3	2	4

	2	1	4	3
$e_2(x)$	4	5	7	8
$t_2(x)$	2	1	5	3

	1	3	2	4
$e_3(x)$	1	6	7	8
$t_3(x)$	1	2	3	6

Figure 5.1: Sample dynamic energy and times functions sorted in non-decreasing order of dynamic energy consumption.

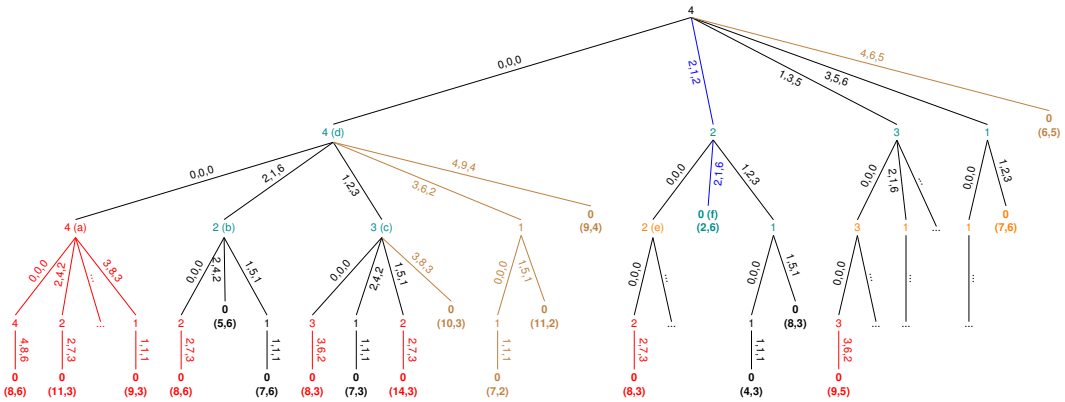

Figure 5.2: The solution tree explored by the naive algorithm to find all distributions and its Pareto-optimal set for a workload $n = 4$ on four processors.

given workload $n = 4$. The input to *HEPOPTA* is four discrete dynamic energy functions, $E = \{e_0(x), \cdots, e_3(x)\}$, as well as four discrete time functions, $T = \{t_0(x), \cdots, t_3(x)\}$, shown in Figure 5.1, with the cardinality of 4 ($m = 4$). They are samples representative of dynamic energy and performance profiles of real-life data-parallel applications. The functions are sorted by dynamic energy in non-decreasing order.

To find the Pareto-optimal solutions for dynamic energy and performance, a straightforward approach is to explore full solution tree and find all possible workload distributions. Figure 5.2 shows the tree, which is constructed by such a naive algorithm. Due to the lack of space, we only show the tree partially.

The tree consist of 4 levels $\{L_0, L_1, L_2, L_3\}$ where all problem sizes given to processor P_i are examined in level L_i. Each node in L_i, $i \in \{0, 1, 2, 3\}$,

is labelled by a positive value representing the workload that is distributed between processors $\{P_i, \cdots, P_3\}$. Each edge connecting a node at the level L_i to its ancestor is labelled by a triple (w, e, t) where w is the problem size assigned to P_i, along with its dynamic energy consumption ($e = e_i(w)$) and its execution time ($t = t_i(w)$).

The exploration process begins from the root to find all distributions for the workload 4 between four processors $\{P_0, P_1, P_2, P_3\}$. Five problem sizes, including all data points in the function $e_0(x)$ and a zero problem size, are assigned to the processor P_0 one after another. Although there is no ordering assumption, we examine the problem sizes in this example in non-decreasing order of their dynamic energy consumption. Assigning the problem sizes $\{0, 2, 1, 3, 4\}$ to P_0 expands the root into 5 children at L_1 representing the remaining workload to be distributed between processors $\{P_1, P_2, P_3\}$. For instance, the edge $(2, 1, 2)$, highlighted in blue in Figure 5.2, indicates that a problem size 2 with a dynamic energy consumption of 1 and an execution time of 2 is given to P_0, and its child is labelled by 2 which equals the remaining size distributed at the level L_1.

In the same manner, each node in levels $\{L_1, L_2, L_3\}$ are expanded towards the leaves. Any leaf node, labelled by 0, illustrates a solution that its dynamic energy consumption is the summation of dynamic energy consumptions and its execution time is the maximum execution times labelling the edges in the path from the root to the leaf. For example, the blue path $\{(2, 1, 2), (2, 1, 6)\}$ in the tree highlights a solution distributing the workload 4 on two processors P_0 and P_1 where its dynamic energy consumption is $2\ (= 1 + 1)$, and its execution time equals $6\ (= \max\{2, 6\})$. It is obvious that the other two processors $\{P_2, P_3\}$ are assigned a zero problem size.

Due to lack of space, we have not shown the branches that do not provide any solution. In a non-solution branch, the summation of problem sizes labelling the edges from the root to its leaf is greater than 4.

In this example, each internal node in the solution tree has either 5 children (or $m + 1$ in general case) or just one child in which case the child is always a leaf. There are two types of leaves: *solution* leaves, labelled by 0 along with its dynamic energy consumption and execution time beneath it, and *no-solution*

leaves, eliminated from, and therefore, not shown in the tree. Each internal node at level L_i, labelled by a positive number w, becomes a root of a solution tree for distribution of the workload w between processors $\{P_i, \cdots, P_3\}$ and is therefore constructed recursively.

Once a solution is found, the algorithm updates the Pareto-optimal set. In the end, the globally Pareto-optimal set includes three members, $\{(\langle 2, 6\rangle, \{2, 2, 0, 0\}), (\langle 4, 3\rangle, \{2, 1, 0, 1\}), (\langle 5, 2\rangle, \{2, 0, 2, 0\})\}$, where each element, like $(\langle eng, eTime\rangle, \{x_0, \cdots, x_3\})$, in the set determines the dynamic energy consumption (eng) and the execution time ($eTime$) of the workload distribution $\{x_0, \cdots, x_3\}$.

The naive algorithm has exponential complexity. We propose *HEPOPTA* which is an efficient recursive algorithm to determine the globally Pareto-optimal set of solutions for data-parallel applications executing on heterogeneous processors. It has polynomial computational complexity. The algorithm shrinks the search space by utilizing three optimizations to avoid exploring whole subtrees in the solution tree.

We will now explain how *HEPOPTA* efficiently solves the aforementioned example. It scans dynamic energy functions, starting with $e_0(x)$, from left to right in non-decreasing order of dynamic energy consumption. The first optimization concerns the upper bound for dynamic energy consumption, which we call it *energy threshold* represented by ε. It is the dynamic energy consumption of the workload distribution which optimizes the execution time of the workload 4 on the processors. We determine this optimal distribution by using *HPOPTA*, an algorithm finding optimal workload distributions minimizing the execution time (Chapter 3). We then initialize the variable ε to the dynamic energy consumption of this distribution. Applying energy threshold enables *HEPOPTA* to shrink search space by ignoring all data points with consumed dynamic energies greater than ε. In the example, the optimal workload distribution, returned by *HPOPTA*, is $X_{t_{opt}} = \{2, 0, 2, 0\}$ with an execution time (t_{opt}) of 2. Therefore, ε in this example is set to 5, which is the dynamic energy consumption for the distribution ($\sum_{i=0}^{p-1} e_i(x_{t_{opt}}[i]) = 5$). *HEPOPTA*, as shown in Figure 5.3, ignores all data points whose dynamic energy consumptions are greater than 5. We highlight in brown all nodes and branches eliminated from

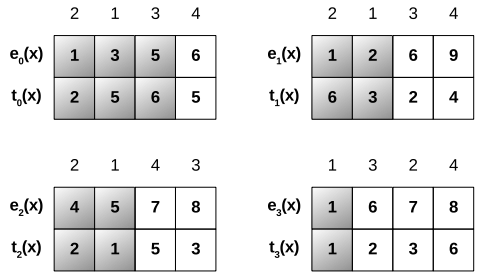

Figure 5.3: Removing some data points from the search space by applying the energy threshold ε.

the solution tree by deploying energy threshold in Figure 5.2. There may exist more than one workload distribution minimizing the execution time but with different dynamic energy consumptions. It is obvious that the best solution is the distribution which minimizes ε. Nevertheless, using a non-optimal ε does not restrain *HEPOPTA* from obtaining the globally Pareto-optimal set.

To shrink the search space further, *HEPOPTA* assigns each level of the tree a size threshold $\sigma_i, i \in \{0, \ldots, p-1\}$. It represents the maximum workload which can be executed in parallel on processors $\{P_i, \cdots, P_{p-1}\}$ so that the dynamic energy consumption of each processor in $\{P_i, \cdots, P_{p-1}\}$ is not greater than ε. In this example, the size threshold vector σ contains four elements, $\sigma = \{\sigma_0, \sigma_1, \sigma_2, \sigma_3\} = \{8, 5, 3, 1\}$. Before expanding each node, *HEPOPTA* compares its workload with its corresponding size threshold. If the workload exceeds the size threshold, the node is not expanded since it results in a solution with a dynamic energy consumption greater than ε.

After calculating the energy threshold ε and the size threshold vector σ, *HEPOPTA* explores the solution tree from its root in the left-to-right and depth-first order. It, first, allocates zero problem sizes to P_0 and P_1 (Figure 5.2). The remaining workload at the level L_2 is 4 which is labelled by $4(a)$ in the tree. Since the workload 4 is greater than the corresponding size threshold σ_2, the node is not expanded further and is cut. This optimization is called operation *Cut*. We highlight in red all sub-trees eliminated from the search space using the operation *Cut*.

Returning to the tree exploration, *HEPOPTA* examines the next node $2(b)$

at the level L_2. Expansion of this node results in two solutions partitioning workload 2 on processors P_2 and P_3. *HEPOPTA* updates the Pareto-optimal set for this node and saves the solution in memory called $PMem$.

HEPOPTA memorizes solutions for each node in levels $\{L_1, \cdots, L_{p-2}\}$. The information stored for a node with a workload of w at a given level L_i, $i \in \{1, \cdots, p-2\}$, is a quintuple $< eng, time, part, P\#, key >$ where eng is the dynamic energy consumption of the solution, $time$ is its parallel execution time on processors $\{P_i, \cdots, P_{p-1}\}$, $part$ is the problem size given to P_i, $P\#$ is the number of active processors in the solution and finally, key, is set to the dynamic energy consumption of a saved Pareto-optimal solution for workload $w - c$ at level L_{i+1}. We call this Pareto-optimal solution at level L_{i+1} a *partial solution* for the workload w. This partial solution may not exist for some nodes, where in this case we represent it by $\emptyset$. Since dynamic energies are unique in a Pareto-optimal set, we use key as a pointer to partial solutions. For each solution leaf in levels $\{L_1, \cdots, L_{p-2}\}$, like $0(f)$ in Figure 5.2, *HEPOPTA* memorizes a solution $\{< 0, 0, 0, 0, \emptyset >\}$.

Thus, the information saved for the node $2(b)$ is a Pareto-optimal set including two members, $\{< 4, 2, 2, 1, \emptyset >, < 6, 1, 1, 2, \emptyset >\}$. We call this key operation, *SavePareto*. Green nodes in the solution tree highlight ones whose Pareto-optimal sets are saved. After $2(b)$, the node $3(c)$ is examined. The solution saved for this node is $\{< 5, 2, 2, 2, \emptyset >\}$.

HEPOPTA then backtracks to the node $4(d)$ on L_1 and builds its Pareto-optimal set by merging Pareto-optimal sets saved for its children, $2(b)$ and $3(c)$. Consider the edge $(2, 1, 6)$ connecting the node $4(d)$ to $2(b)$. Merging this edge with the Pareto-optimal set which has been already saved for $2(b)$, $\{< 4, 2, 2, 1, \emptyset >, < 6, 1, 1, 2, \emptyset >\}$, results in one Pareto-optimal solution for the node $4(d)$ which is saved as the quintuple $< 5, 6, 2, 2, \mathbf{4} >$. In this solution, the last element , 4, which is highlighted in bold, points to its partial solution in the node $2(b)$ at L_2, which is $\{< 4, 2, 2, 1, \emptyset >\}$. Merging the edge $(1, 2, 3)$ with the Pareto-optimal set for $3(c)$, $\{< 5, 2, 2, 2, \emptyset >\}$, results in a new solution $\{< 7, 3, 1, 3, 5 >\}$. Therefore, the Pareto-optimal set for the node $4(d)$ is $\{< 7, 3, 1, 3, 5 >, < 5, 6, 2, 2, 4 >\}$, which is saved in the memory.

After building and saving the Pareto-optimal set of the node $4(d)$, *HEP-*

OPTA visits the node $2(e)$ at the level L_2. This node has already been explored, and therefore, its Pareto-optimal set is retrieved from $PMem$. We call this key operation, *ReadParetoMem*. The nodes whose solutions are retrieved from the memory are highlighted in orange.

After visiting the other remaining nodes, *HEPOPTA* backtracks to the root and builds the globally Pareto-optimal solutions for the workload 4 executing on processors $\{P_0, \cdots, P_3\}$ using the Pareto-optimal sets saved for its children. Then it terminates.

HEPOPTA deploys three key operations, including a). *Cut*, b). *SavePareto*, and c). *ReadParetoMem*, to efficiently explore solution trees and build globally Pareto-optimal solutions optimizing for dynamic energy and performance. We now describe the pseudocode of the proposed algorithm using these key operations as the fundamental building blocks.

Similar to the two optimization algorithms *HPOPTA* and *HEOPTA*, proposed in Chapters 3 and 4 respectively, *HEPOPTA* is a branch-and-bound algorithm. However, it determines upper bounding criteria in a different way. In addition, the key operation *SavePareto* saves partial Pareto-optimal solutions, in comparison with the key operation *Save* in *HPOPTA* and *HEOPTA*, which memorizes optimal workload distributions.

5.3 Formal Description of *HEPOPTA*

We present the pseudocode of *HEPOPTA* in Algorithm 5. The Inputs of the algorithm are: the problem size, n, the number of heterogeneous processors, p, an array of p dynamic energy profiles, $E = \{E_0, E_1, \cdots, E_{p-1}\}$ and p time functions $T = \{T_0, T_1, \cdots, T_{p-1}\}$, where E_i is the dynamic energy function, and T_i represents the execution time of processor P_i, $i \in \{0, \cdots, p-1\}$. Each energy function comprises m pairs (x_{ij}, e_{ij}), $j \in \{0, 1, \cdots, m-1\}$, so that x_{ij} is the j-th problem size in the function and e_{ij} represents the amount of dynamic energy consumed by running it on P_i. Each time function includes m pairs (x_{ij}, t_{ij}), $j \in \{0, 1, \cdots, m-1\}$, so that x_{ij} is the j-th problem size in the function and t_{ij} represents its execution time on P_i. *HEPOPTA* returns Ψ_{EP},

the globally Pareto-optimal solutions. It consists of a set where each element of the set is a triple like $(eng, time, X)$. The first field eng is the dynamic energy consumption of a Pareto-optimal solution, $time$ represents its execution time, and $X = \{x_0, x_1, \cdots, x_{p-1}\}$ determines the workload distribution of the solution. The solutions are sorted in increasing order of dynamic energy.

HEPOPTA starts by sorting energy and time functions in non-decreasing order of dynamic energy consumption and execution time, respectively (Line 2). Both original and sorted functions are kept. Original functions are assumed to be sorted by problem size. Then, HPOPTA is invoked to find the optimal distribution minimizing the execution time of the workload n on p processors (Line 3). This function returns the optimal execution time, t_{opt}, along with its distribution, $X_{t_{opt}}$. The energy threshold ε is initialised to the dynamic energy consumption of the distribution $X_{t_{opt}}$ (Line 4). The function READFUNC(E_i, x) returns the dynamic energy consumption of the problem size x executing on the processor P_i. It returns 0 when x is equal to 0.

The size threshold array σ is initialised by using the function SIZETHRESHOLDCALC (Line 5). A 2D array $PMem$, with dimensions of $(p-2) \times (n+1)$, is defined to save Pareto-optimal solutions for processors $\{P_1, \cdots, P_{p-2}\}$, which are found during the tree exploration (Line 6). Then, HEPOPTA_KERNEL is invoked to explore the solution tree and determines the globally Pareto-optimal set of solutions for dynamic energy and performance, returned in Ψ_{EP}.

Although *HEPOPTA* is a one-dimensional data-partitioning algorithm, it can be directly used to solve problems with two or three dimensions through reducing the dimensionality to 1D. The detail has been explained in Chapter 3.

The pseudocodes of all functions as well as the structure of $PMem$ are described in Appendix D.

5.3.1 Recursive Algorithm $HEPOPTA\_Kernel$

Algorithm 6 shows the pseudocode for *HEPOPTA_Kernel*. It efficiently explores the solution tree and recursively builds Pareto-optimal solutions from tree leaves to the root. Pareto-optimal solutions for a given node at level L_i, $i \in \{0, 1, \cdots, p-2\}$, are built by merging all solutions stored for its children,

Algorithm 5 Algorithm Finding Globally Pareto-optimal Solutions for Dynamic Energy and Performance for a Workload n on p Heterogeneous Processors.

1: **function** HEPOPTA(n, p, E, T, Ψ_{EP})
INPUT:
Problem size, $n \in \mathbb{Z}_{>0}$
Number of processors, $p \in \mathbb{Z}_{>0}$
Dynamic energy profiles, $E = \{E_0, ..., E_{p-1}\}$,
$E_i = \{(x_{ij}, e_{ij}) \mid i \in [0, p), j \in [0, m), x_{ij} \in \mathbb{Z}_{>0}, e_{ij} \in \mathbb{R}_{>0}\}$.
Time functions, $T = \{T_0, ..., T_{p-1}\}$,
$T_i = \{(x_{ij}, t_{ij}) \mid i \in [0, p), j \in [0, m), x_{ij} \in \mathbb{Z}_{>0}, t_{ij} \in \mathbb{R}_{>0}\}$.
OUTPUT:
Pareto-optimal solutions for dynamic energy and performance, Ψ_{EP},
$\Psi_{EP} = \{(eng_k, time_k, X_k) \mid k \in [0, |\Psi_{EP}|)\}$,
$X_k = \{x_k[0], x_k[1], \cdots, x_k[p-1]\}$,
$x_k[i] \in \{\bigcup_{j=0}^{m-1} x_{ij} \cup \{0\}\},\ i \in [0, p)$.

2: $E \leftarrow E \cup Sort_{\uparrow}(E)$, $T \leftarrow T \cup Sort_{\uparrow}(T)$
3: $(X_{t_{opt}}, t_{opt}) \leftarrow$ HPOPTA(n, p, T)
4: $\varepsilon \leftarrow \sum_{i=0}^{p-1}$ READFUNC($E_i, x_{t_{opt}}[i]$)
5: $\sigma \leftarrow$ SIZETHRESHOLDCALC(p, E, ε)
6: $PMem[i][j] \leftarrow \emptyset, \forall i \in \{1, \cdots, p-2\}, j \in \{0, \cdots, n\}$
7: HEPOPTA_KERNEL($n, p, 0, E, T, \varepsilon, \sigma, X_{cur}, PMem, \Psi_{EP}$)
8: **return** Ψ_{EP}
9: **end function**

placed at level L_{i+1}. *HEPOPTA_Kernel* uses three operations *Cut*, *Save-Pareto* and *ReadParetoMem*, illustrated in Section 5.2, to reduce the search space and achieve a polynomial computational complexity.

The variable $c \in \{0, \cdots, p-1\}$ indicates the tree level that is processing in the current recursion of *HEPOPTA_Kernel*. Prior to expanding a node at the level L_c, *HEPOPTA_Kernel* determines whether its workload exceeds σ_c. If it is the case then the node is not explored (Lines 2-4). Lines 5-10 process solutions found at the last level L_{p-1}. If there exists a solution, the function returns $TRUE$, otherwise $FALSE$.

Before exploring a node at a given level c, $c \in \{L_1, \cdots, L_{p-2}\}$, the function READPARETOMEM is called to retrieve from $PMem$ the solution set saved for the current workload n on processors $\{P_c, \cdots, P_{p-1}\}$ (Lines 11-18). The variable *status* determines the type of retrieved solutions. If no solution is already stored for the node or the total dynamic energy consumption of all the retrieved solutions is greater than or equal to ε (given by the status, *NOT_SOLUTION*), *HEPOPTA_Kernel* returns $FALSE$ and backtracks. If at least one of the solutions, in the retrieved set, has a total dynamic energy consumption less than ε (given by the status, *SOLUTION*), the function returns $TRUE$. If none of

the above cases happen, the routine starts expanding the node by initializing pointer idx to -1 and $x_{c\ idx}$ to 0 (Lines 19-35). The variable idx, ranging from -1 to $m-1$, determines indexes of data points in the functions, and $x_{c\ idx}$ represents the problem size of idx-th data point in the functions.

The $while$ loop (Lines 22-35) examines all data points with dynamic energy consumption less than or equal to ε in the function E_c, sorted in non-decreasing order of energy consumption. The array $X_{cur} = \{x_{cur}[0], \cdots, x_{cur}[p-1]\}$, where $x_{cur}[i] \in \{\bigcup_{j=0}^{m-1} x_{ij} \cup \{0\}\}$, stores problem sizes currently assigned to processors $P_i (i \in \{0, 1, \cdots, p-1\})$. In each iteration, the data point idx is extracted from E_c, and its problem size ($x_{c\ idx}$) is stored in array $x_{cur}[c]$ (Line 24). *HEPOPTA_Kernel* is recursively invoked to find solutions for the remaining workload $n - x_{c\ idx}$ at the next level L_{c+1} (Line 25). If there exists any solution for the workload, $x_{c\ idx}$ is added to $partsVec$, a list holding all problem sizes, given to P_c, which result in Pareto-optimal solutions (Lines 26-29).

If idx reaches the end of the energy profile E_c, the *while* loop terminates (Lines 31-33), otherwise, idx is incremented to examine the next data point in the energy profile E_c.

After exploring all children of the current node, the function MERGEPARTIALPARETOES is invoked to merge and store the Pareto-optimal solutions of its children into a single Pareto-optimal set of solutions.

In the end, the corresponding memory cell storing the Pareto-optimal solution for a node with a workload n at L_c ($PMem[c][n]$) is labelled *Finalized* (Line 39). Finalizing a memory cell implies that this cell contains the final solutions. *HEPOPTA_Kernel* returns $TRUE$ provided that exploring the node, processed in the current recursion, has led to a solution (Line 40).

Algorithm 6 Recursive Kernel Invoked by Function $HEPOPTA$

1: **function** HEPOPTA_KERNEL($n, p, c, E, T, \varepsilon, \sigma, X_{cur}, PMem, \Psi_{EP}$)
2: **if** CUT(n, σ_c) **then**
3: **return** $FALSE$
4: **end if**
5: **if** $c = p - 1 \wedge$ READFUNC(E_c, n) $\leq \varepsilon$ **then**
6: $x_{cur}[c] \leftarrow n$
7: **return** $TRUE$
8: **else**
9: **return** $FALSE$
10: **end if**
11: **if** $n \neq 0 \wedge c \geq 1 \wedge c \leq p - 2$ **then**
12: $status \leftarrow$ READPARETOMEM($n, c, \varepsilon, PMem$)
13: **if** $status = NOT\_SOLUTION$ **then**
14: **return** $FALSE$
15: **else if** $status = SOLUTION$ **then**
16: **return** $TRUE$
17: **end if**
18: **end if**
19: $idx \leftarrow -1$; $x_{c\ idx} \leftarrow 0$
20: $isSol \leftarrow FALSE$
21: $partsVec \leftarrow \varnothing$
22: **while** READFUNC($E_c, x_{c\ idx}$) $\leq \varepsilon$ **do**
23: **if** $x_{c\ idx} \leq n$ **then**
24: $x_{cur}[c] \leftarrow x_{c\ idx}$
25: $outRes \leftarrow$ HEPOPTA_KERNEL($n - x_{c\ idx}, p, c + 1, E, T, \varepsilon, \sigma, X_{cur}, PMem, \Psi_{EP}$)
26: **if** $outRes = TRUE$ **then**
27: $isSol \leftarrow TRUE$
28: $partsVec \leftarrow partsVec \cup x_{c\ idx}$
29: **end if**
30: **end if**
31: **if** $n = 0 \vee idx + 1 = m$ **then**
32: **break**
33: **end if**
34: $idx \leftarrow idx + 1$
35: **end while**
36: **if** $c \geq 1 \wedge c \leq p - 2$ **then**
37: MERGEPARTIALPARETOES($n, p, c, E, T, partsVec, PMem, \Psi_{EP}$)
38: **end if**
39: MAKEPARETOFINAL($PMem[c][n]$)
40: **return** $isSol$
41: **end function**

5.4 *HTPOPTA*: Algorithm Finding Globally Pareto-optimal Solutions for Total Energy and Performance

An effective approach to save energy in green computing clusters, big data centres and web servers is to switch idle nodes off or to put them in the sleep mode [57, 58, 9, 59, 60]. It reduces the base energy of idle nodes that eventually results in less total energy consumption of the platform. However, applying a hybrid approach which minimizes the total energy consumption of active nodes and puts idle nodes in sleep mode or turns them off can reduce the total energy consumption of the whole platform more effectively. For this, each node consumes energy optimally during its computations and is switched off at other times. As a result, the total energy consumption of the system can be reduced significantly.

We will describe here the solution to the bi-objective optimization problem of data parallel applications for total energy and performance, which we call *HTPOPT*.

The problem is formulated as follows: Given a problem size n running on p heterogeneous processors, whose dynamic energy and performance functions are respectively represented by $E = \{e_0(x), ..., e_{p-1}(x)\}$ and $T = \{t_0(x), ..., t_{p-1}(x)\}$, and P_S is the base power of the platform.

Eq. 5.2 formulates the bi-objective optimization problem to obtain workload distributions minimizing execution time and total energy consumption during the parallel execution of the workload n using the p processors.

$$
\begin{aligned}
&HTPOPT(n, p, m, T, E, P_S) : \\
&\min_{X} \quad \{\max_{i=0}^{p-1} t_i(x_i), P_S \times \max_{i=0}^{p-1} t_i(x_i) + \sum_{i=0}^{p-1} e_i(x_i)\} \\
&\text{Subject to:} \\
&\sum_{i=0}^{p-1} x_i = n \\
&0 \leq x_i \leq m, \qquad i = 0, \cdots, p-1 \\
&\text{where} \quad p, n, m \in \mathbb{Z}_{>0}, \quad x_i \in \mathbb{Z}_{\geq 0}, \quad t_i(x), e_i(x), P_S \in \mathbb{R}_{\geq 0}
\end{aligned}
\tag{5.2}
$$

We prove that the solution to the problem *HTPOPT* is a subset of the globally Pareto-optimal set of solutions for dynamic energy and execution time determined by the algorithm *HEPOPTA*. The correctness proof is presented in Appendix E.

We propose an algorithm called *HTPOPTA* (**H**eterogeneous **T**otal energy-**P**erformance **OPT**imization **A**lgorithm) solving *HTPOPT*.

5.5 Formal Description of *HTPOPTA*

The function *HTPOPTA* calculates globally Pareto-optimal solutions for total energy and performance using Ψ_{EP}. It takes as input the problem size, n, the number of heterogeneous processors, p, an array of p dynamic energy functions, $E = \{E_0, E_1, \cdots, E_{p-1}\}$, an array of p time functions $T = \{T_0, T_1, \cdots, T_{p-1}\}$ and the base power of the platform, P_S. *HTPOPTA* returns the globally Pareto-optimal set for execution time and total energy which are stored in Ψ_{TP}. It is a set of triples like $(teng, time, X)$ where $teng$ illustrates the total energy consumption of a Pareto-optimal solution, $time$ is its execution time, and $X = \{x_0, x_1, \cdots, x_{p-1}\}$ represents the workload distribution of the solution.

HTPOPTA, first, calls HEPOPTA to find globally Pareto-optimal solutions for dynamic energy and performance, Ψ_{EP} (Line 2). It then calculates the total

energy consumption of every solution in Ψ_{EP} (Line 4) and enquiries if there exists a solution in Ψ_{TP} where its total energy consumption is equal to that of the new solution but with less execution time or with the same execution times but less active processors. If this is the case, the current solution in Ψ_{TP} is updated by the new one (Lines 5-15). Otherwise, the new solution is added into Ψ_{TP} (Line 17).

After inserting solutions, non-Pareto-optimal solutions are found (Lines 22-28) to get eliminated from Ψ_{TP} (Line 29). Pareto-optimal solutions in Ψ_{TP} are also sorted in the increasing order of total energy consumption and decreasing order of execution time. It should be noted that solutions in Ψ_{EP} and Ψ_{TP} are sorted in increasing order of energy consumption, and consequently in decreasing order of execution time.

5.6 Experimental Results

In this section, we experimentally study the practical performance of *HEPOPTA* and *HTPOPTA*.

All the experiments are conducted on two heterogeneous hybrid servers, *HCLServer01* and *HCLServer02*. *HCLServer01* consists of one Intel Haswell CPU, one Nvidia K40c GPU and one Intel Xeon Phi 3120P, whose specifications can be found in Table 5.1. It involves three abstract processors CPU_1, GPU_1 and Phi_1. *HCLServer02* contains an Intel Skylake multicore CPU and one Nvidia P100 PCIe GPU, whose specifications are explained in Table 5.2. The abstract processors for this node are CPU_2 and GPU_2. We earlier explained how to model a hybrid platform as a set of loosely-coupled abstract processors in chapter 3.

We use two data-parallel applications, Matrix Multiplication and 2D discrete Fourier transform. Each application utilizes highly optimized vendor specific kernels for the CPUs and the accelerators. The Matrix Multiplication application (DGEMM) uses Intel MKL DGEMM [143] for CPUs, ZZGEMMOOC out-of-card package [144] for Nvidia GPUs and XeonPhiOOC out-of-card package [146] for Intel Xeon Phis. ZZGEMMOOC and XeonPhiOOC packages respec-

Algorithm 7 Algorithm Finding Globally Pareto-optimal Solutions for Total Energy and Performance using *HEPOPTA*

function $HTPOPTA(n, p, E, T, P_S, \Psi_{TP})$
INPUT:
Problem size, $n \in \mathbb{Z}_{>0}$
Number of processors, $p \in \mathbb{Z}_{>0}$
Energy profiles, $E = \{E_0, ..., E_{p-1}\}$,
$E_i = \{(x_{ij}, e_{ij}) \mid i \in [0, p), j \in [0, m), x_{ij} \in \mathbb{Z}_{>0}, e_{ij} \in \mathbb{R}_{>0}\}$.
Time functions, $T = \{T_0, ..., T_{p-1}\}$,
$T_i = \{(x_{ij}, t_{ij}) \mid i \in [0, p), j \in [0, m), x_{ij} \in \mathbb{Z}_{>0}, t_{ij} \in \mathbb{R}_{>0}\}$.
Base power of the heterogeneous platform, $P_S \in \mathbb{R}_{>0}$
OUTPUT:
Pareto-optimal solutions for total energy and performance, Ψ_{TP},
$\Psi_{TP} = \{(teng_k, time_k, X_k) \mid k \in [0, |\Psi_{TP}|)\}$,
$X_k = \{x_k[0], x_k[1], \cdots, x_k[p-1]\}$,
$x_k[i] \in \{\bigcup_{j=0}^{m-1} x_{ij} \cup \{0\}\},\ i \in [0, p)$.

HEPOPTA(n, p, E, T, Ψ_{EP})
for all $tup \in \Psi_{EP}$ **do**
  $te \leftarrow tup.eng + P_S \times tup.time$
  $tup' \leftarrow \{x \mid x \in \Psi_{TP}, x.eng = te\}$
  if $tup' \neq \emptyset$ **then**
    if $tup.time < tup'.time$ **then**
      $tup' \leftarrow (te, tup.time, tup.X)$
    else if $tup.time = tup'.time$ **then**
      $idle_{tup} \leftarrow \{x | x \in tup.X, x = 0\}$
      $idle_{tup'} \leftarrow \{x | x \in tup'.X, x = 0\}$
      if $|idle_{tup}| < |idle_{tup'}|$ **then**
        $tup' \leftarrow (te, tup.time, tup.X)$
      end if
    end if
  else
    $\Psi_{TP} \leftarrow \Psi_{TP} \cup (te, tup.time, tup.X)$
  end if
end for
$minTime \leftarrow \infty$
$nPList \leftarrow \emptyset$
for all $tup \in \Psi_{TP}$ **do**
  if $tup.time \geq minTime$ **then**
    $nPList \leftarrow nPList \cup tup$
  else
    $minTime \leftarrow tup.time$
  end if
end for
return $(\Psi_{TP} - nPList)$
end function

Table 5.1: HCLServer01: Specifications of the Intel Haswell multicore CPU, Nvidia K40c, and Intel Xeon Phi 3120P.

Intel Haswell E5-2670V3	
No. of cores per socket	12
Socket(s)	2
CPU MHz	1200.402
L1d cache, L1i cache	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30720 KB
Total main memory	64 GB DDR4
Memory bandwidth	68 GB/sec
NVIDIA K40c	
No. of processor cores	2880
Total board memory	12 GB GDDR5
L2 cache size	1536 KB
Memory bandwidth	288 GB/sec
Intel Xeon Phi 3120P	
No. of processor cores	57
Total main memory	6 GB GDDR5
Memory bandwidth	240 GB/sec

Table 5.2: HCLServer02: Specifications of the Intel Skylake multicore CPU and Nvidia P100 PCIe.

Intel Xeon Gold 6152	
Socket(s)	1
Cores per socket	22
L1d cache, L1i cache	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30976 KB
Main memory	96 GB
NVIDIA P100 PCIe	
No. of processor cores	3584
Total board memory	12 GB CoWoS HBM2
Memory bandwidth	549 GB/sec

tively reuse CUBLAS and MKL BLAS for in-card DGEMM calls. In Chapter 6, we will explain that the out-of-card packages allow the accelerators to run computations for any arbitrary size. In the 2D FFT application, Intel MKL FFT [147] for CPUs and Xeon Phis, and CUFFT [148] for Nvidia GPUs are deployed. Unlike the Matrix Multiplication application, all computations for the FFT application are in-card. The Intel MKL and CUDA versions used on *HCLServer01* are respectively 2017.0.2 and 7.5, and on *HCLServer02* are 2017.0.2 and 9.2.148.

The Matrix Multiplication application computes $C = \alpha \times A \times B + \beta \times C$, where A, B, and C are matrices of size $m \times n$, $n \times n$, and $m \times n$, respectively and α and β are constant floating-point numbers. Workloads range from 64×10112 to 28800×10112 with a step size of 64 for the first dimension m. The speed for the execution of a given problem size $m \times n$ is calculated as $\frac{2 \times m \times n^2}{t}$ where t is execution time taken to compute the result matrix C.

Figures 5.4 and 5.5 show DGEMM speed and dynamic energy functions of the five abstract processors. To highlight variations in performance profiles, Figure 5.4 presents zoomed speed functions for *CPU_1*, *GPU_1* and *CPU_2*, where the width of variations reaches 22%. In addition, the zoomed figure highlights that the speed of *GPU_1* starts growing after a 10% drop in performance. We can see the same variations in the *Phi_1* profile as well.

The 2D FFT application computes 2D FFT of a complex signal matrix of size $m \times n$. Workloads range from 1024×51200 to 10000×51200 with a step size 16 for the first dimension m. The experimental data set does not include problem sizes which cannot be factored into primes less than or equal to 127. For these problem sizes, CUFFT for GPU gives failures. The speed for the execution of a given problem size $m \times n$ is calculated as $\frac{m \times n \times \log_2(m \times n)}{t}$ where t is execution time taken to compute 2D FFT of size $m \times n$.

Figures 5.6 and 5.7 show the speed and dynamic energy functions of CPU_1, GPU_1, Phi_1, CPU_2 and GPU_2 abstract processors. Phi_1 consumes ten times more dynamic energy than the other processors. Figure 5.8 excludes the dynamic energy function of Phi_1 to highlight variations in the dynamic energy profiles of CPU_1, GPU_1, CPU_2 and GPU_2.

The performance and the dynamic energy functions are built separately

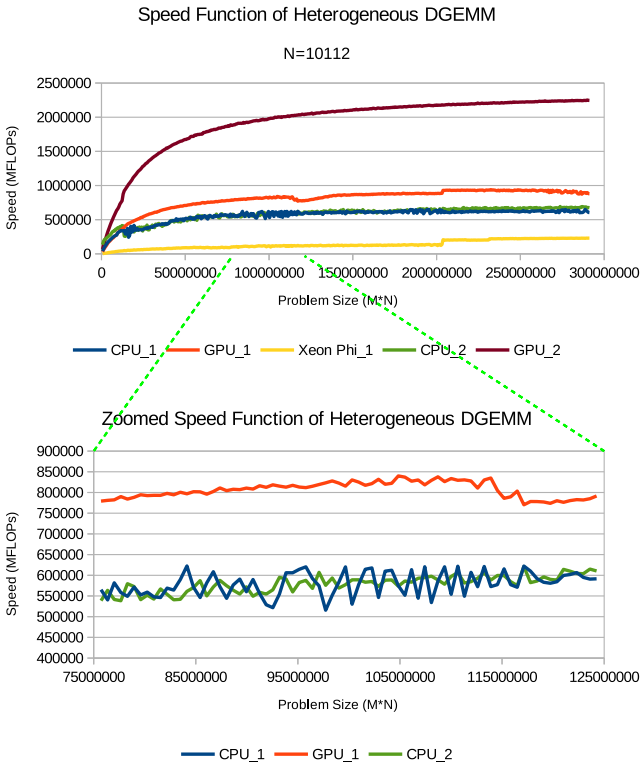

Figure 5.4: Full and zoomed speed functions of the heterogeneous Matrix Multiplication application executing on HCLServer01 *and* HCLServer02.

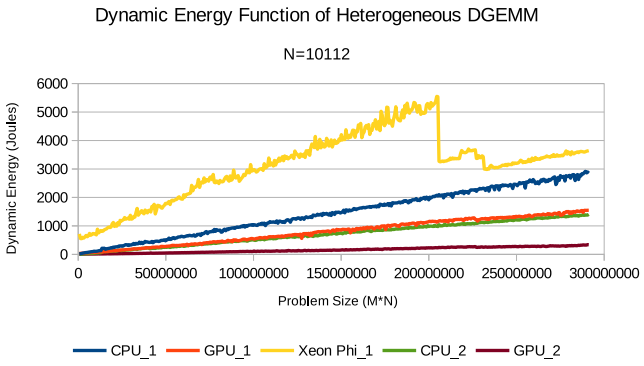

Figure 5.5: Dynamic energy functions of heterogeneous Matrix Multiplication application executing on HCLServer01 *and* HCLServer02.

experimentally, as explained in Chapters 3 and 4. To ensure the reliability of the experimental results, we follow a detailed statistical methodology which

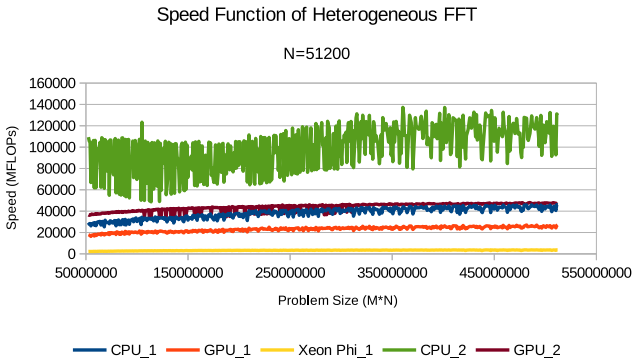

Figure 5.6: Speed functions of the heterogeneous 2D FFT application executing on HCLServer01 *and* HCLServer02*.*

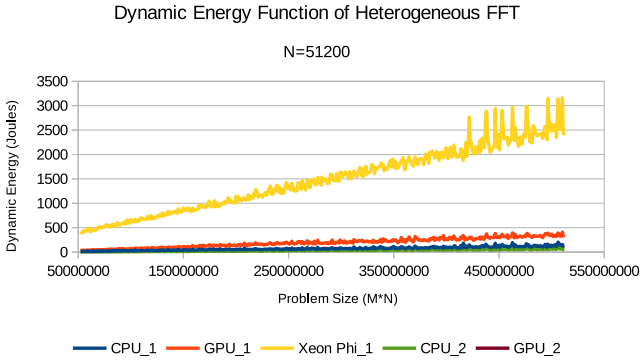

Figure 5.7: Dynamic energy functions of the heterogeneous 2D FFT application executing on HCLServer01 *and* HCLServer02*.*

we explain in Appendix A. Briefly, to obtain a data point for each function, the software follows Student's t-test method and executes the application repeatedly until the sample means of the measurement (execution time\dynamic energy\total energy) lies in user-defined confidence interval and a user-defined precision is achieved. We set the confidence interval as 95% and the precision as 10% for our experiments.

5.6.1 Analysis of *HEPOPTA*

We create an experimental data set for all experiments including all the problem sizes in discrete functions for performance and dynamic energy.

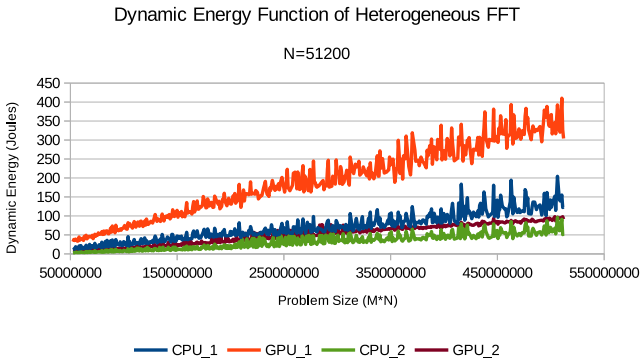

Figure 5.8: Dynamic energy functions of the heterogeneous 2D FFT application executing on HCLServer01 *and* HCLServer02*. In this figure, the dynamic energy profile for Phi_1 is ignored.*

The experimental data set for DGEMM is $\{64 \times 10112, 128 \times 10112, 196 \times 10112, \cdots, 57600 \times 10112\}$, and for FFT is $\{1024 \times 51200, 1040 \times 51200, 1056 \times 51200, \cdots, 20000 \times 51200\}$.

For the first set of our experiments, we determine the minimum, average and maximum cardinality of globally Pareto-optimal sets determined by *HEPOPTA*. These values for the Matrix Multiplication application are $(1,\ 55,\ 96)$, and for the 2D FFT application, $(1,\ 11,\ 33)$. If the cardinality of the globally Pareto-optimal set for an input problem size is 1, the output workload distribution optimizes both performance and dynamic energy consumption. Globally Pareto-optimal sets with the maximum cardinality for Matrix Multiplication and FFT are shown in the Figures 5.9 and 5.10. In these figures, the blue point above Pareto-optimal sets represents the execution time and dynamic energy consumption of the load-balanced distribution.

We study improvements in performance and reductions in the dynamic energy consumption of optimal solutions determined by *HEPOPTA* in comparison with load-balanced workload distribution for the second set of the experiments. A load balance distribution is one with the minimum difference between the execution times of processors. The number of active processors in load-balanced solutions may be less than the total number of processors ($p = 5$). The percentage of performance improvement is obtained using $Perf\ Improvement(\%) = \frac{t_{balance} - t_{opt}}{t_{opt}} \times 100$, where $t_{balance}$ represents the

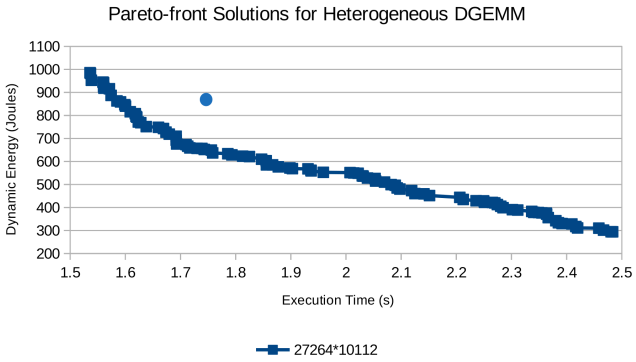

Figure 5.9: Globally Pareto-front solutions for dynamic energy and execution time with the maximum cardinality determined by HEPOPTA *for the heterogeneous Matrix Multiplication application.*

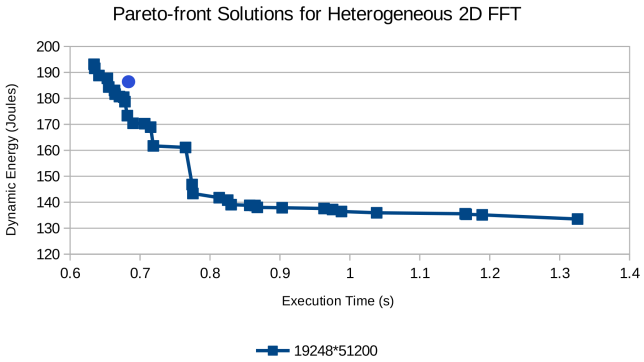

Figure 5.10: Globally Pareto-front solutions for dynamic energy and execution time with the maximum cardinality determined by HEPOPTA *for the heterogeneous 2D FFT application.*

execution time of the load balance distribution, and t_{opt} is the optimal execution time. For Matrix Multiplication, the average and maximum performance improvements are 26% and 102% respectively, and for 2D FFT are respectively 7% and 44%. The percentage of dynamic energy saving is calculated as $Energy\ Saving(\%) = \frac{e_{balance} - e_{opt}}{e_{opt}} \times 100$, where $e_{balance}$ represent the dynamic energy consumption of load balance distribution, and e_{opt} is optimal dynamic energy consumption. The average and maximum energy saving for are found to be respectively 130% and 257% for the Matrix Multiplication application, and 44% and 105% for the 2D FFT.

Table 5.3: Percentage improvement in performance when the dynamic energy consumption is increased by up to 5% over the optimal one on HCLServer01 *and* HCLServer02*.*

Application	Average	Max
DGEMM	5%	50%
FFT	19%	109%

Table 5.4: Percentage reduction in dynamic energy consumption by 5% degradation in performance over the optimal distribution on HCLServer01 *and* HCLServer02*.*

Application	Average	Max
DGEMM	18%	116%
FFT	6%	63%

Finally, we obtain to what extent performance can be improved when the dynamic energy consumption is increased by up to 5% over the optimal one. The average and maximum performance gains will be reported. The percentage of performance improvement is calculated using $Perf\ Improvement(\%) = \frac{t_{e_{opt}} - t_{(e_{opt} \times 1.05)}}{t_{(e_{opt} \times 1.05)}} \times 100$, where $t_{e_{opt}}$ and $t_{(e_{opt} \times 1.05)}$ are respectively the execution time of the energy-optimal endpoint and execution time associated with 5% increase in energy consumption over the optimal.

Table 5.3 summarizes the average and maximum percentage of performance improvement for the Matrix Multiplication and 2D FFT applications.

We also determine the average and maximum energy savings by 5% degradation in performance over the optimal distribution. The dynamic energy saving is obtained using $Energy\ Saving(\%) = \frac{e_{t_{opt}} - e_{(t_{opt} \times 1.05)}}{e_{(t_{opt} \times 1.05)}} \times 100$, where $e_{t_{opt}}$ and $e_{(t_{opt} \times 1.05)}$ are respectively the dynamic energy consumption of the performance-optimal endpoint in the Pareto-optimal front and the dynamic energy consumption associated with 5% degrade in performance over the optimal.

The average and maximum percentage of dynamic energy saving for the applications are shown in Table 5.4.

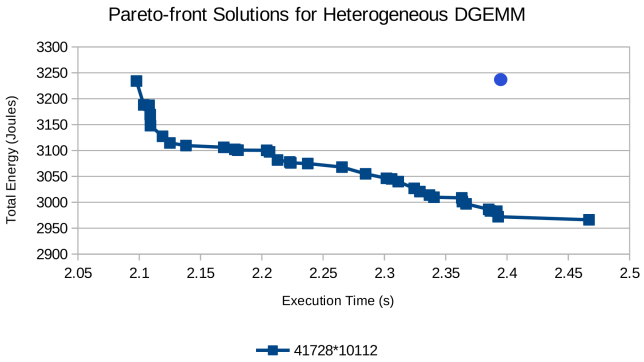

Figure 5.11: Globally Pareto-front solutions for total energy and execution time with the maximum cardinality determined by HTPOPTA *for the heterogeneous Matrix Multiplication application.*

5.6.2 Analysis of *HTPOPTA*

We use the same experimental data sets, as those employed for analysis of *HEPOPTA*, to conduct our three sets of experiments for *HTPOPTA*.

First, the minimum, average and maximum cardinality of globally Pareto-optimal sets for total energy and execution time is studied. These values for the Matrix Multiplication application are $(1, 15, 35)$, and for the 2D FFT application are $(1, 2, 8)$. The cardinalities are less than the corresponding values for the globally Pareto-optimal sets for dynamic energy and execution time since the Pareto-optimal set for total energy and execution time is a subset of Pareto-optimal set for dynamic energy and execution time. Globally Pareto-optimal sets of total energy consumption and execution time with the maximum cardinality for Matrix Multiplication and FFT are shown in Figures 5.11 and 5.12. In Figure 5.11, the point above the Pareto-optimal solutions represents the execution time and total energy consumption of load-balanced distribution. The load-balanced solutions in Figure 5.12 have not been shown because of being far away from the sets.

We then study the compromise between execution time and total energy consumption. We calculate how much performance can be gained in case the total energy consumption is increased by up to 5% over the optimal one. The average and maximum percentage of performance speed-ups are illustrated

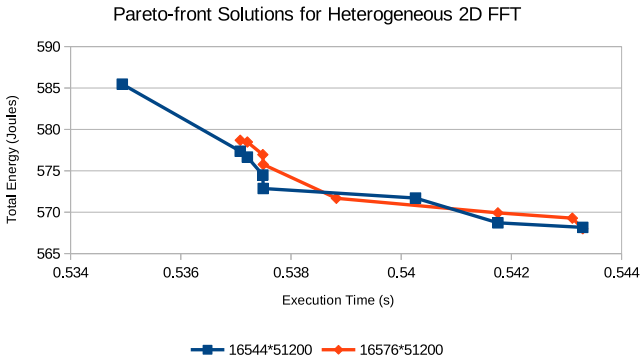

Figure 5.12: Globally Pareto-front solutions for total energy and execution time with the maximum cardinality determined by HTPOPTA *for the heterogeneous 2D FFT application. Each curve represents a problem size.*

Table 5.5: Percentage improvements in total energy consumption by 5% *increase of total energy consumption over the optimal distribution on* HCLServer01 *and* HCLServer02*.*

Application	Average	Max
DGEMM	8%	17%
FFT	0.7%	9%

in Table 5.5. The percentage of performance improvement is obtained using $Perf\ Improvement(\%) = \frac{t_{te_{opt}} - t_{(te_{opt} \times 1.05)}}{t_{(te_{opt} \times 1.05)}} \times 100$, where $t_{te_{opt}}$ and $t_{(te_{opt} \times 1.05)}$ are respectively the execution time of the total energy-optimal endpoint and execution time associated with 5% increase in total energy consumption over the optimal.

We also determine the average and maximum total energy savings by 5% degradation in performance over the optimal. The energy saving is obtained using $TEnergy\ Saving(\%) = \frac{te_{t_{opt}} - te_{(t_{opt} \times 1.05)}}{te_{(t_{opt} \times 1.05)}} \times 100$, where $te_{t_{opt}}$ and $te_{(t_{opt} \times 1.05)}$ are respectively the total energy consumption of the performance-optimal endpoint in the Pareto-optimal front and the total energy consumption associated with 5% degrade in performance over the optimal. Table 5.6 summarizes the experimental results for the Matrix Multiplication and 2D FFT applications.

Using *HEOPTA*, one can find workload distributions minimizing dynamic energy consumption. *HTPOPTA* provides workload distributions which min-

Table 5.6: Percentage total energy saving when performance is degraded by up to 5% over the optimal one on HCLServer01 *and* HCLServer02*.*

Application	Average	Max
DGEMM	4%	13%
FFT	0.4%	6%

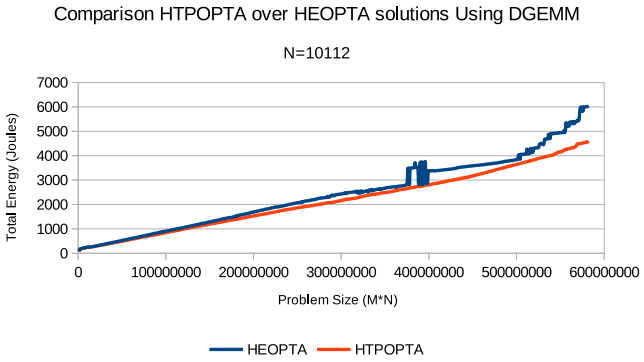

Figure 5.13: Total energy profiles of the heterogeneous Matrix Multiplication application for two different workload distributions HTPOPTA *and* HEOPTA *executing on* HCLServer01 *and* HCLServer02*.*

imize total energy consumption. To demonstrate that dynamic energy optimization does not always result in minimizing total energy, we calculate the percentage total energy saving over *HEOPTA* solutions for the aforementioned data set. Total energy saving is calculated as follows: $Total\ Energy\ Saving = \frac{te_{HEOPTA} - te_{opt}}{te_{opt}} \times 100$, where te_{HEOPTA} is total energy consumption of the solution with optimal dynamic energy consumption and te_{opt} is the optimal total energy consumption. The minimum, average and maximum total energy savings for the DGEMM application are 0, 11% and 37%, respectively. Zero percentage total energy saving represents that the same workload distribution is determined by *HTPOPTA* and *HEOPTA*. Figure 5.13 shows the optimal total energy consumption of the Matrix Multiplication application over the total energy consumption of the distribution with minimum dynamic energy consumption.

The minimum, average and maximum total energy savings for the 2D FFT application are respectively 0, 29% and 106%. Figure 5.14 compares 2D FFT

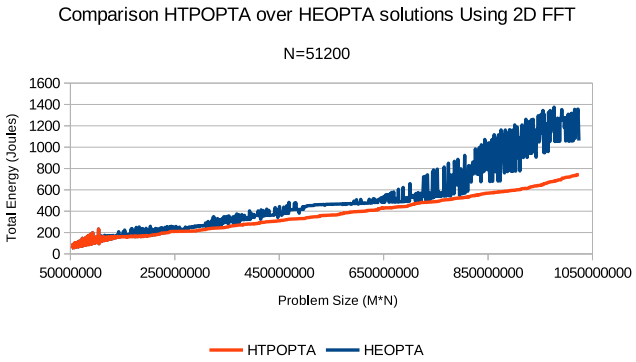

Figure 5.14: Total energy profiles of the heterogeneous 2D FFT application for two different workload distributions HTPOPTA *and* HEOPTA *executing on* HCLServer01 *and* HCLServer02*.*

optimal total energy consumption over the total energy consumption of the distribution with minimum dynamic energy consumption.

5.7 Summary

Performance and energy are now the most dominant objectives for optimization on modern heterogeneous HPC platforms such as computational clusters, supercomputers and cloud computing infrastructures. Recent research efforts on modern multicore platforms demonstrate that the performance and dynamic energy profiles of data-parallel applications executing on such platforms exhibit drastic variations due to inherent complexities in these platforms such as severe resource contention for shared resources (such as Last Level Cache (LLC), interconnects, PCI-E links, etc.) and Non-Uniform Memory Access (NUMA). Due to these variations, these works show that the discrete functional relationships between performance and workload size and between energy and workload size have non-linear and non-convex shapes thereby demonstrating that the workload distribution has become an important decision variable that can no longer be ignored. There are algorithms solving the problem of bi-objective optimization of data-parallel applications for performance and dynamic energy on *homogeneous* HPC platforms (BOPPE) where

use workload distribution as the sole decision variable.

We presented in this chapter algorithms to solve two bi-objective optimization problems for performance and energy on *heterogeneous* HPC platforms. The first optimization problem, *HEPOPT*, has two objectives, execution time and dynamic energy, and one decision variable, the *workload distribution*. We proposed a novel data partitioning algorithm called *HEPOPTA* solving *HEPOPT*. Its inputs include the problem size, n, the number of available heterogeneous processors, p, p discrete performance functions (one for each processor) and p discrete dynamic energy functions (one for each processor). *HEPOPTA* returns the globally Pareto-optimal solutions for performance and dynamic energy within a polynomial complexity of $O(m^3 \times p^3 \times \log_2(m \times p))$, where m represents the cardinality of the discrete performance and dynamic energy functions.

The decision variable of the second optimization problem, *HTPOPT*, is the same, *workload distribution*, but the objectives are execution time and *total* energy. It was proved that *HEPOPTA* can be reused to solve *HTPOPT*.

We experimentally analysed the scalability and efficiency of the algorithms using two heterogeneous data-parallel applications, matrix multiplication and two-dimensional discrete fast Fourier transform on a hybrid cluster of two heterogeneous nodes.

Regarding the experimental results, one can conclude that the globally Pareto-optimal front of solutions contains the best load-balanced solutions. Our algorithms therefore determine better Pareto-optimal front of load-imbalanced solutions that are totally ignored by load-balancing approaches.

We empirically demonstrated that dynamic energy optimization does not always lead to the minimization of total energy consumption. The average and maximum difference in total energy consumption between the dynamic-energy optimal and total-energy optimal solutions is (11%,37%) for matrix multiplication, and (29%,106%) for 2D FFT.

The software implementations for *HEPOPTA* and *HTPOPTA* are available at [158].

Chapter 6

Out-of-card Implementation for Accelerator Kernels on Heterogeneous Computing Platforms

In this chapter, we focus on the implementation of a programming interface for out-of-card kernels on heterogeneous HPC platforms. We describe a library, which is called *HCLOOC*, containing interfaces that address inherent challenges, such as the limited main memory of accelerators and limited bandwidth of the PCI-E communication links, connecting accelerators to host processors. It employs optimal software pipelines to overlap data transfers between the host CPU and accelerators with computations on the accelerators. It is designed using the fundamental building blocks, which are OpenCL command queues for FPGAs, Intel offload streams for Xeon Phis, and CUDA streams and events that allow concurrent utilization of the copy and execution engines provided in Nvidia GPUs.

Experimental results show that the proposed out-of-card implementation achieves 85% of the peak double-precision floating performance of Nvidia P100 PCIe GPU and a speedup of 6 times over the Nvidia's out-of-card matrix multiplication implementation (CUBLAS-XT). It will also be demonstrated that

our implementation exhibits 0% drop in performance when the problem size exceeds the main memory of the GPU. We observe this 0% drop also for our implementations for Intel Xeon Phi and Xilinx FPGA.

6.1 Introduction to Out-of-card Computation for Accelerators

Extreme-scale high performance computing (HPC), big data platforms, and clouds today feature hybrid nodes containing multicore CPU processors and one or more accelerators such as GPUs, Intel Xeon Phis and FPGAs to facilitate execution of workloads that demand high energy efficiency.

Despite high performance and superior performance per watt, hardware acceleration in HPC poses some prominent challenges on efficient utilization of accelerators to solve big instances of data-parallel applications on hybrid nodes. These challenges are summarized below:

1. **Limited memory size of accelerators:** Accelerators typically have smaller main memory compared to that of the host multicore CPU connected to it. Consider the Top500 list of supercomputers [8]. Both Summit and Sierra supercomputers, respectively ranked first and second, are composed of IBM POWER9 multicore CPUs, which support 1024 GB per socket, and Nvidia Volta GV100 accelerators, which provides only 16 GB main memory. The forth-ranking Tianhe-2A supercomputer includes Intel Ivy bridge multicore CPUs, with 768 GB main memory per socket, and Intel Xeon Phi 31S1P accelerator, which provides only 8 GB main memory. The AI Bridging Cloud Infrastructure (ABCI), currently holding the seventh place in Top500 rating, employs Intel Xeon Gold CPUs, which can support 768 GB main memory, and Nvidia Tesla V100 SXM2 accelerators, which support up to 16 GB of main memory. Table 6.1 shows a list of Nvidia GPUs, launched in 2018, along with their main memory sizes. It is apparent that the main memory capacity of accelerators is far less than that of host CPUs. Since all data accessed by a given kernel

should be transferred into the accelerator prior to any kernel invocation, the maximum problem size, which can be solved by an accelerator, is consequently limited by its main memory capacity. Therefore, to execute large problem sizes of an application using accelerators, it is required that out-of-card implementations of the application are either available or developed from scratch.

2. **Limited bandwidth of the PCI-E communication link:** Out-of-card executions usually entail multiple data transfers of data structures (that fit inside the main memory of the accelerator) from the host CPU to the accelerator and back. Accelerators are connected to CPUs using PCI-E communication links. However, the limited bandwidth of these links impacts the execution times of out-of-card implementations. In addition, accelerators such as GPUs provide advanced hardware support to facilitate overlap of data transfers between host and device and computations on the device. For example, modern Nvidia GPUs (K40, K80, P100, etc) provide three engines: two copy engines, one for host-to-device transfers and another for device-to-host transfers, and a kernel engine. Therefore, libraries aiming to provide efficient out-of-card implementations must utilize the vendor-supplied optimizations to overlap the communications over PCI-E communication links with the computations on accelerators.

3. **Lack of efficient libraries for out-of-card computations:** There is an abysmal lack of libraries providing interfaces that allow programmers to write efficient out-of-card implementations for their data-parallel kernels on accelerators. There are exceptions (but very few) such as Nvidia's CUBLAS-XT package [136], which provides a set of BLAS routines that utilize multiple GPUs and MAGMA [141], which provides out-of-card dense LU, Cholesky and QR factorizations. Victream [159] is a directed acyclic graph (DAG) computing framework for out-of-card computations on multiple GPUs. However, from our experiments, it is observed that vendor out-of-card implementations (such as CUBLAS-XT [136]) are not the best in terms of performance.

Table 6.1: List of Nvidia GPUs launched in 2018 with their main memory capacities.

Model	Memory Size
GeForce GT 1030	2 GB
GeForce GTX 1050	3 GB
GeForce GTX 1060	6 GB
GeForce RTX 2080 Ti	11 GB
Tesla T4	16 GB
TITAN V-CEO Edition	32 GB

In this chapter, we present a library (*HCLOOC*) that address these challenges and allows programmers to write efficient out-of-card implementations of data-parallel kernels for mainstream accelerators such as GPUs, Xeon Phis and FPGAs. The library is a wrapper that reuses the fundamental building blocks such as OpenCL command queues [160] for FPGAs, Intel offload streams [161] for Intel Xeon Phis, and CUDA streams and events that allow concurrent utilization of the copy and execution engines provided in Nvidia GPUs [137], [162].

The library contains two principal components. The first component, *Partitioner*, partitions a workload into blocks where each block can fit into the accelerator's main memory. The second component, *Stream Engine*, uses a configurable software pipeline to overlap data transfers from host CPU to the accelerator and back and kernel invocations in the accelerator. This component reuses the vendor-supplied optimization engines for the data transfers. For example, for the out-of-card implementation of matrix multiplication that we present in this section, the *Stream Engine* uses a five-stage pipeline.

Our proposed interface, *HCLOOC*, is evaluated by the implementation of out-of-card matrix multiplications for Nvidia GPUs, Intel Xeon Phis and FPGAs. We also demonstrate that it outperforms CUBLAS-XT, an out-of-card BLAS package implemented by Nvidia.

6.2 Out-of-card Library for Accelerator Kernels (*HCLOOC*)

In this section, we present our library, called *HCLOOC*, which allows programmers to write out-of-card implementations for their kernels on accelerators such as GPUs, Xeon Phis and FPGAs. *HCLOOC* consists of two principal components:

- **Partitioner:** It allows partitioning of input and output data structures into partitions that fit into an accelerator's main memory.
- **Stream Engine:** It uses a configurable software pipeline to overlap data transfers from host CPU to the accelerator and back and invocations of in-card kernels in the accelerator. It is a wrapper that utilizes the fundamental building blocks such as OpenCL command queues [160] for FPGAs, Intel offload streams [161] for Intel Xeon Phis, and CUDA streams that allow concurrent utilization of the copy and execution engines provided in Nvidia GPUs [137], [162].

We consider a simple example, matrix multiplication, to illustrate our partitioner and the structure of software pipeline employed in *HCLOOC*. The *Stream Engine* uses a five-stage pipeline for the out-of-card implementation of matrix multiplication that we present in this section.

6.2.1 Implementation for Dense Matrix Multiplication on a GPU using *HCLOOC*

In this section, we elucidate the core logic in the two components (*Partitioner* and *Stream Engine*) by describing our out-of-card implementation of matrix multiplication of large dense matrices on Nvidia GPUs.

The implementation computes $C = \alpha \times A \times B + \beta \times C$, where A, B, and C are matrices of size $M \times K$, $K \times N$, and $M \times N$, respectively and α and β are constant floating-point numbers. If the workload size ($M \times K + K \times N + M \times N$) fits into the memory of GPU, all three matrices are transferred to the device,

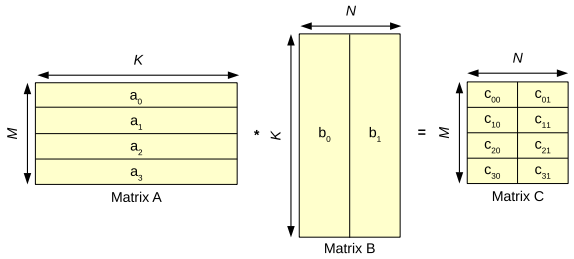

Figure 6.1: Employing Partitioner *module to decompose matrix A into 4 horizontal slices, matrix B into 2 vertical slices, and matrix C into 8 ($= 4 \times 2$) blocks.*

the kernel CUBLAS DGEMM [145] is then invoked to update matrix C, and the resultant matrix C is returned to the host. But, when the workload size exceeds the main memory of the accelerator, the data transfer of the matrices will fail.

The first step in our out-of-card implementation is to partition the three matrices A, B and C. *Partitioner* splits matrix A into h equal horizontal slices, matrix B into v equal vertical slices, and matrix C into $h \times v$ equal rectangular blocks ensuring that the data required for updating any two blocks of C in the same column is small enough to fit in the accelerator's memory.

For example, suppose M, N and K to be 4, 4 and 8 respectively, resulting in the total workload size equal to 80 ($4 \times 8 + 8 \times 4 + 4 \times 4$) matrix elements. Suppose the GPU's main memory can only store 44 matrix elements. Then *Partitioner* will be applied, and it will split matrix A into 4 horizontal slices, matrix B into 2 vertical slices and consequently matrix C into eight 1×2 blocks guaranteeing that the data required for updating of any two blocks of C in the same column will fit in the memory of the accelerator. Figure 6.1 shows the matrix decomposition. Although other decompositions are possible (for example, partitioning A, B and C into 2, 4 and 8 sub-blocks respectively), *Partitioner* will return the decomposition, which additionally optimizes the work of the target software pipeline. In this particular case, the pipeline uses two sets of buffers and two parallel streams, and in order to optimize the use of the resources, *Partitioner* is instructed to select the decomposition with the smallest possible v.

Stream Engine is then employed to execute the out-of-card implementation. CUDA streams and asynchronous communications are used to optimally utilize concurrent access of copy and execution engines provided in Nvidia GPUs thereby achieving optimal overlapping of communication with computation.

The columns of blocks C are computed one after the other. In each column, the blocks are computed going from the top to the bottom. Each iteration is associated with multiple transfers of the matrix blocks between the host memory and device memory, which leads to a significant communication cost. To reduce the communication overhead, *HCLOOC* overlaps data transfers and kernel invocations. To achieve this, two sets of data buffers are allocated in the GPU's main memory. Each set is used for updating one block of C. While one block of C is being updated, the required data for the second block of C is transferred into the second set of buffers.

Stream Engine uses a five-stage software pipeline to execute the out-of-card implementation. Figure 6.2 presents the pipeline structure for three matrices A, B and C decomposed in the Figure 6.1. The stages of pipeline are described as following:

- **S(b_i):** Sending the i-th slice of matrix B (i.e. b_i) from host to device.
- **S(a_i):** Sending the i-th slice of matrix A (i.e. a_i) from host to device.
- **S(c_{ij}):** Sending a rectangular block of matrix C (i.e. c_{ij}) from host to device.
- **DGEMM:** Vendor-supplied optimized DGEMM (CUBLAS) invocation.
- **R(c_{ij}):** Sending the updated block c_{ij} of C back from device to host.

Since blocks on matrix C are updated in the column order, the first stage of the pipeline (S(b_i)) occurs every h step (h is the number of horizontal slices). The stream, which updates c_{ij}, transfers horizontal slice a_i, vertical slice b_j (if it has not already been transferred into the accelerator memory), and block c_{ij} from host to the accelerator memory. After updating c_{ij} by invoking in-card CUBLAS DGEMM, it is sent back to the host. In figure 6.2, it is supposed

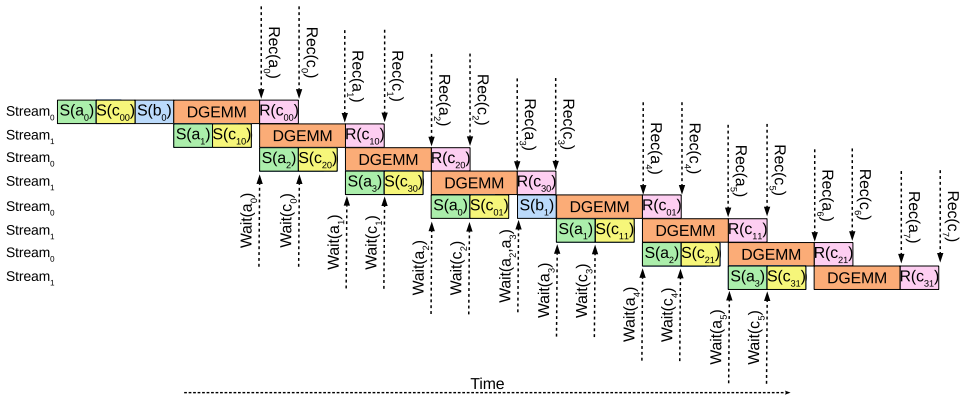

Figure 6.2: Pipeline structure in Stream Engine *module for sample matrices shown in Figure 6.1 on a GPU with dual copy engines and one execution engine which supports concurrent data transfers in two directions (represented by $S()$ calls) and overlapping of data transfers and kernel executions (represented as DGEMM). Events, $Rec(x)$ and $Wait(x)$, are used for synchronization of data transfers.*

that GPU is provided with dual copy engines, which supports concurrent data transfers in two directions. Since creating a new stream has some overhead, we exploit and reuse just two streams in a round robin order so that while one stream is involved in doing computation, the other is transferring.

To make sure data stored in device buffers will not be overwritten until kernel executions that operate on the data have completed, we create events for each sub-matrix existing in A and C. As shown in figure 6.2, $Rec(x)$ represents recording the event associated with a block x, and $Wait(x)$ makes the process wait for the event associated with the block x until it is recorded.

We have explained *HCLOOC* using dense matrix multiplication which is one of level-3 BLAS routines. To implement out-of-card matrix-vector operations (level-2 BLAS routines) using *HCLOOC*, matrix A is partitioned. Vector b is completely transferred to the device and stored from start to the completion of the out-of-card operation. Vector c is also partitioned and updated during the course of the out-of-card operation. In our future work, we will consider triangular and banded matrices.

Stream Engine: Further Details

Stream Engine is responsible for transferring input data from host CPU to GPU, invocations of in-card *CUBLAS_DGEMM* [145], an implementation of BLAS on top of the Nvidia CUDA runtime, and transferring the resultant output blocks back to the host.

Since out-of-card computation is associated with lots of data transfer between host CPU and the accelerator, we use two sets of buffers on the GPU which include 5 buffers. Two out of 5 buffers, $dA[0]$ and $dA[1]$, store two slices of A, one buffer, dB, stores one slice of B and the remaining two buffers, $dC[0]$ and $dC[1]$, are used for sub-blocks of C. Employing two sets of buffers along with two CUDA streams enables communication-computation overlapping on the accelerator.

Algorithm 8 illustrates the work of *Stream Engine* for Nvidia GPUs. Inputs to the module are matrices A, B and C, with sizes of $M \times K$, $K \times N$ and $M \times N$ respectively, and h and v which are determined by *Partitioner* (section 6.2.1). There exist two CUDA streams (Line 2). Operations issued into a stream are executed in issue-order, while operations submitted to different streams can be executed concurrently. Since the GPU supports concurrent copy and execution engines, the designed out-of-card matrix multiplication implementation utilizes concurrent data transfers in both directions. For synchronization of data communications, we create two sets of CUDA event arrays (Line 3).

At the beginning, slices a_0, b_0 and block $c_{0,0}$ are transferred into device buffers $dA[0]$, dB and $dC[0]$ (Lines 9-13). While $dC[set]$, $set = \{0, 1\}$, is being updated by in-card CUBLAS_DGEMM kernel (line 15), next sub-matrices of A, C (Lines 17-22) and following sub-matrix of B (if it is applicable) (Lines 23-27) are asynchronously transferred to the device by the other stream. After finishing its computation, the current stream records $event_a$ to release buffer dA (Line 16). Line 28 is responsible for sending the result back to the host. Then, the current stream releases its dC (Line 29) to be reused by the other stream. Finally, the last block, $c_{(h-1)(j-1)}$, is updated and sent back to the host memory (Lines 31, 32).

The amount of communication that could be overlapped with the computa-

tion depends on the ratio of the communication time and the computation time. If the CUBLAS_DGEMM execution dominates the total execution time, then, the smaller the ratio, the more communication could be overlapped. However, if the communication over the PCI-E bus dominates the total execution time, the communication would always be a bottleneck.

Algorithm 8 *Stream Engine* using CUDA Streams and Events to Execute Out-of-card DGEMM Implementation

function *Stream Engine*(A, B, C, M, N, K, h, v)
  Stream stream[2]
  Event $event_a[h * v]$, $event_c[h * v]$
  for $j = 0; j < v; j++$ **do**
    for $i = 0; i < h; i++$ **do**
      $idx \leftarrow i + j * h$
      $set \leftarrow idx \ \% \ 2$, $set\_ \leftarrow (idx + 1) \ \% \ 2$
      $i\_ \leftarrow (idx + 1) \ \% \ h$, $j\_ \leftarrow \frac{idx+1}{h}$
      if $idx = 0$ **then**
        MEMCPYASYNC($b_0 \rightarrow dB, stream[idx\%2]$)
        MEMCPYASYNC($a_0 \rightarrow dA[set], stream[idx\%2]$)
        MEMCPYASYNC($c_{0,0} \rightarrow dC[set], stream[idx\%2]$)
      end if
      if $idx < (h * v - 1)$ **then**
        CUBLAS_DGEMM($dA[set], dB, dC[set], stream[idx\%2]$)
        EVENTRECORD($event_a[idx], stream[idx\%2]$)
        if $idx > 0$ **then**
          STREAMWAITEVENT($stream[(idx + 1)\%2], event_a[idx - 1]$)
          MEMCPYASYNC($a_{i\_} \rightarrow dA[set\_], stream[(idx + 1)\%2]$)
          STREAMWAITEVENT($stream[(idx + 1)\%2], event_c[idx - 1]$)
          MEMCPYASYNC($c_{i\_,j\_} \rightarrow dC[set\_], stream[(idx + 1)\%4]$))
        end if
        if $i = (h - 1) \wedge j < (v - 1)$ **then**
          STREAMWAITEVENT($stream[(idx + 1)\%2], event_a[idx]$)
          STREAMWAITEVENT($stream[(idx + 1)\%2], event_a[idx - 1]$)
          MEMCPYASYNC($b_{j+1} \rightarrow dB, stream[(idx + 1)\%2]$))
        end if
        MEMCPYASYNC($dC[set] \rightarrow c_{i,j}, stream[idx\%2]$)
        EVENTRECORD($event_c[idx], stream[idx\%2]$)
      else
        CUBLAS_DGEMM($dA[set], dB, dC[set], stream[idx\%2]$)
        MEMCPYASYNC($dC[set] \rightarrow c_{i,j}, stream[idx\%2]$)
      end if
    end for
  end for
end function

We have elucidated the principal components of *HCLOOC* by implementing out-of-card dense matrix multiplication for Nvidia GPUs. For out-of-card implementations for Xeon Phis and FPGAs, *Stream Engine* uses Intel offload streams (for Xeon Phis) and OpenCL command queues (for FPGAs). Computational kernels for Xeon Phis and FPGAs would be vendor-optimized BLAS library routine DGEMM.

Partitioner: Further Details

Partitioner decomposes matrix A into h horizontal slices, a_i ($0 \leq i \leq h-1$), B into v vertical slices, b_j ($0 \leq j \leq v-1$), and matrix C consequently into $h * v$ blocks, $c_{i,j}$. Partitioning of the matrices is performed such that certain constraints and optimization criteria are satisfied:

- Each matrix is decomposed into sub-matrices of approximately the same size. This ensures load balancing and maximum concurrency.
- GPU's main memory is divided between 5 buffers organized into two sets: $dA[0]$ and $dC[0]$ in one set, $dA[1]$ and $dC[1]$ in the other set. dB is shared between the sets.
- The number of slices in matrix B should be as small as possible. There is only one buffer on the accelerator for matrix B. Since the buffer is shared between two streams, data transfer from CPU to GPU cannot be overlapped with computation for matrix B slices, and this degrades the communication-computation overlap in the pipeline structure. Therefore, *Partitioner* decomposes matrices so to minimize the number of slices of B.
- Minimizing the number of slices in B may result in too many slices in A. We have experimentally found that the performance of *HCLOOC* degrades when matrix A is partitioned into too many slices. To prevent this, *Partitioner* minimizes the product: $h \times v$.

Algorithm 9 shows the core logic of *Partitioner* in this specific case where the inputs are matrix sizes, M, N and K, and the memory size of the accelerator is mem_size. Outputs are h, v, $heights$ and $widths$. $heights$ is an array of size h where $heights[i]$ contains the number of rows in the i-th slice of matrix A. Similarly, $widths$ is an array of size v where $widths[i]$ contains the number of columns in the i-th slice of matrix B.

We know that the size of $dA[set]$ is $\frac{M}{h} \times K$, dB is $K \times \frac{N}{v}$, and $dC[set]$ is $\frac{M}{h} \times \frac{N}{v}$, where $set = \{0, 1\}$. Since all 5 buffers should fit into the accelerator

memory, buffer sizes must satisfy the following equation (6.1).

$$2 \times \frac{M}{h} \times K + K \times \frac{N}{v} + 2 \times \frac{M}{h} \times \frac{N}{v} = mem\_size \tag{6.1}$$

From Eq. 6.1, we derive the following expression for v:

$$v = \frac{2 \times M \times N + N \times K \times h}{mem\_size \times h - 2 \times M \times K} \tag{6.2}$$

The valid values for h are $\{2, 3, \cdots, M\}$. Algorithm 9 initializes h to 2, and v is then calculated using the formula 6.2 (Lines 3). Then the number of horizontal slices is increased (h_{temp}) until the best decomposition is achieved, which minimizes the number of slices in matrix B (Lines 5-14). Finally, rows of matrix A are distributed amongst h slices, and columns of matrix B are distributed amongst v slices (Lines 18-21).

Algorithm 9 Decomposition of Matrices A, B, and C using the *Partitioner*

1: **function** PARTITIONER(M, N, K, mem_size)
2: $h \leftarrow 2$
3: $v \leftarrow \lceil \frac{2 \times M \times N + N \times K \times h}{mem\_size \times h - 2 \times M \times K} \rceil$
4: $h_{temp} \leftarrow 3$, $v_{temp} \leftarrow \infty$
5: **while** $h_{temp} \leq M \wedge (v_{temp} \leq 0 \vee v_{temp} > 1)$ **do**
6: $v_{temp} \leftarrow \lceil \frac{2 \times M \times N + N \times K \times h_{temp}}{mem\_size \times h_{temp} - 2 \times M \times K} \rceil$
7: **if** $v_{temp} > 0$ **then**
8: **if** $v \leq 0 \vee (v_{temp} < v \wedge h \times v \geq h_{temp} \times v_{temp})$ **then**
9: $h \leftarrow h_{temp}$
10: $v \leftarrow v_{temp}$
11: **end if**
12: **end if**
13: $h_{temp} \leftarrow h_{temp} + 1$
14: **end while**
15: **if** $v \leq 0 \vee v > N$ **then**
16: There is no distribution to fit into the accelerator memory, exit.
17: **end if**
18: $heights[i] \leftarrow \frac{M}{h}, i \in [0, h-1]$
19: $widths[i] \leftarrow \frac{N}{v}, i \in [0, v-1]$
20: $heights[i] + +, i \in [0, M\%h)$
21: $widths[i] + +, i \in [0, N\%v)$
22: **end function**

6.3 Experimental Results

In this section, we demonstrate the performance of our out-of-card implementations of matrix multiplication for large dense matrices on GPUs, Xeon Phis and FPGAs. For this purpose, three packages ZZGemmOOC [144], XeonPhiOOC [146] and FPGAOOC [163] have been developed, which use the interfaces defined as *HCLOOC*. We also study the speedup of ZZGemmOOC over Nvidia's out-of-card BLAS package CUBLAS-XT [136].

6.3.1 Evaluation Platform

Our experiments are conducted on two heterogeneous servers, named *HCLServer01* and *HCLServer02*. The first system, *HCLServer01*, contains an Intel Haswell multicore CPU hosting a Nvidia K40c GPU, an Intel Xeon Phi 3120P co-processor and a Virtex 7 690T FPGA whose specifications are shown in Table 6.2. While the host Haswell CPU contains 64 GB main memory, Nvidia K40c GPU has only 12 GB, Intel Xeon Phi has only 6 GB, and Xilinx Virtex 7 690T FPGA involves only 16 GB main memory. In this node, the Nvidia GPU and Intel Xeon Phi communicate with the host CPU using low-bandwidth PCI-E x16 links. The FPGA communicates using PCI-E x8 link. Another platform, *HCLServer02*, includes an Intel Skylake multicore CPU with 96 GB main memory which is integrated with one Nvidia P100 PCIe GPU, containing 12 GB main memory. The Nvidia GPU communicates using PCI-E x16 link. The specification of *HCLServer02* can be found in Table 6.3. Both Nvidia K40c and P100 PCIe GPUs are provided with three engines including dual copy engines and one kernel invocation engine.

6.3.2 Performance of Out-of-card Implementations

We have developed three packages based on *HCLOOC* library which perform out-of-card matrix multiplication of large dense matrices on GPUs, Xeon Phis and FPGAs. For GPU, ZZGemmOOC out-of-card package [144] is developed that reuses CUBLAS for in-card DGEMM invocations. For Xeon Phi, XeonPhiOOC out-of-card package [146] is developed that reuses MKL BLAS [143]

Table 6.2: HCLServer01: Specifications of the Intel Haswell multicore CPU, Nvidia K40c, Intel Xeon Phi 3120P, and Xilinx Virtex 7 690T FPGA.

Intel Haswell E5-2670V3	
No. of cores per socket	12
Socket(s)	2
CPU MHz	1200.402
L1d cache, L1i cache	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30720 KB
Total main memory	64 GB DDR4
Memory bandwidth	68 GB/sec
NVIDIA K40c	
No. of processor cores	2880
Total board memory	12 GB GDDR5
L2 cache size	1536 KB
Memory bandwidth	288 GB/sec
Intel Xeon Phi 3120P	
No. of processor cores	57
Total main memory	6 GB GDDR5
Memory bandwidth	240 GB/sec
Xilinx Virtex 7 690T FPGA	
Frequency	200 MHz
LUTs	693120
DSPs	3600
BRAM	53 MB
FFs	866,400
Total main memory	16 GB DDR3

Table 6.3: HCLServer02: Specifications of the Intel Skylake multicore CPU and Nvidia P100 PCIe.

Intel Xeon Gold 6152	
Socket(s)	1
Cores per socket	22
L1d cache, L1i cache	32 KB, 32 KB
L2 cache, L3 cache	256 KB, 30976 KB
Main memory	96 GB
Nvidia P100 PCIe	
No. of processor cores	3584
Total board memory	12 GB CoWoS HBM2
Memory bandwidth	549 GB/sec

for in-card DGEMM invocations. For FPGA, FPGAOOC out-of-card package [163] is designed that reuses a user-defined kernel for in-card invocations. The user-defined kernel calculates matrix multiplication using the straightforward algorithm with three nested loops. The kernel is not fully optimized for FPGA and just uses work item pipelining. All packages use the interface defined by $HCLOOC$. However, they are different in terms of implementation details. For instance, while *Stream Engine* in ZZGemmOOC package is implemented using CUDA, XeonPhiOOC uses Intel offload streams, and FPGAOOC adapts OpenCL command queues. The Intel MKL and CUDA versions used on *HCLServer01* are respectively $2017.0.2$ and 7.5. The CUDA version $9.2.148$ is installed on *HCLServer02*.

To evaluate the efficiency of our out-of-card implementations, we measure the execution speed of our packages. The speed of multiplication of two matrices with sizes $M \times K$, $K \times N$ is calculated as $\frac{2 \times M \times K \times N}{t}$ where t represents the execution time, which includes the time taken for matrix multiplication and data transfers from host to device and vice versa.

We show the speeds for ZZGemmOOC and XeonPhiOOC for problem sizes in the set, $\{64^2, 128^2, \cdots, 44800^2\}$. Since FPGAOOC is very slow for all problem sizes, we evaluate this package for very small problem sizes. To study the out-of-card computation on the FPGA, the memory size of FPGA is manually set to 64 KB. Matrix sizes used for this implementation is the set,

$\{16^2, 32^2, \cdots, 512^2\}$. In these experiments, when workload size fits into the accelerator memory, all three matrices are transferred to the device, and results are calculated using in-card computations.

To obtain an experimental data point, the application is executed repeatedly until the sample mean lies in the 95% confidence interval and a precision of 0.025 (2.5%) has been achieved. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions using Pearson's chi-squared test. When we mention a single number such as floating-point performance (in GFLOPS), it is assumed that we are referring to the sample mean determined using the Student's t-test.

Figure 6.3 compares the speed functions of ZZGEMOOC with CUBLAS-XT on TESLA K40c GPU. In this figure, x-axis represents the size of square matrices and y-axis represents speed in GFLOPS. It is apparent that ZZGemmOOC outperforms CUBLAS-XT for all work-sizes. The gap between ZZGemmOOC and CUBLAS-XT becomes wider as the matrix size grows. The peak double-precision floating point performance of TESLA K40c is 1.43 TFLOPS. The maximum double-precision floating point performance of ZZGemmOOC is 1.17 TFLOPS, which constitutes 82 percent of the peak. ZZGemmOOC provides $1.5x$ speedup over CUBLAS-XT for small matrix sizes and achieves up to $2.7x$ speedup for larger ones. The vertical green line in the figure separates the results for in-card matrix multiplication and out-of-card matrix multiplication.

Figure 6.4 shows the speed function for XeonPhiOOC on Intel Xeon Phi 3120P. We could not find any good third-party implementation for performance comparison. The green line in the figure separates the results for in-card matrix multiplication and out-of-card matrix multiplication. The peak double-precision floating point performance of Intel Xeon Phi 3120P is 1003 GFLOPS. Using XeonPhiOOC, the maximum double-precision floating point performance is 725 GFLOPS, which constitutes 72 percent of the peak.

Figure 6.5 illustrates the speed function for matrix multiplication on Xilinx Virtex 7 690T FPGA. The green line in the figure highlights the data point where out-of-card computation starts. It is clear that FPGAOOC exhibits no

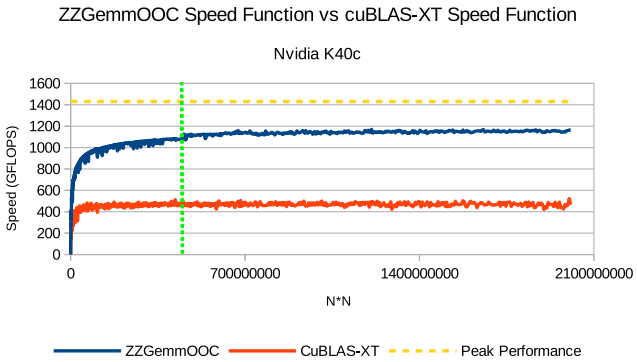

Figure 6.3: Comparison of vendor-optimized library CUBLAS-XT with ZZGemmOOC on Nvidia K40c GPU. The green line separates in-card computations from out-of-card ones. The dotted yellow line represents the theoretical peak double precision performance of the GPU.

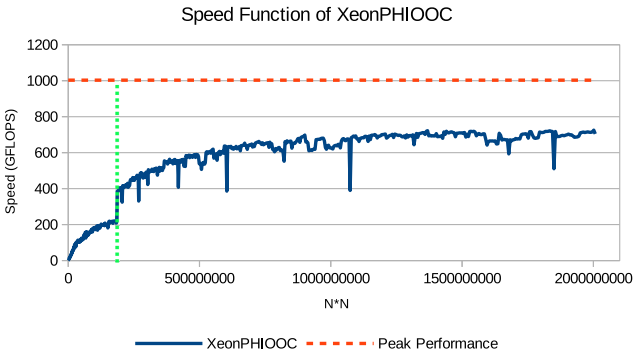

Figure 6.4: Speed function of XeonPhiOOC on Intel Xeon Phi 3120P. The green line separates in-card computations from out-of-card ones. The dotted red line represents the theoretical peak double-precision performance.

drop in speed for out-of-card computations in comparison with in-card results. There exists no third-party implementation that could be used for performance comparison.

The speed functions of ZZGEMOOC is compared with CUBLAS-XT on TESLA P100 PCIe GPU in Figure 6.6. Similar to the out-of-card matrix multiplication on TESLA K40c GPU, ZZGemmOOC outperforms CUBLAS-XT for all workload sizes and the difference between our out-of-card library and CUBLAS-XT becomes wider as the matrix size grows. The peak double-

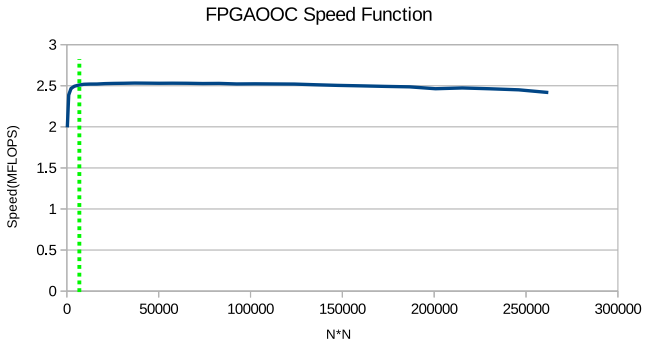

Figure 6.5: Speed function of FPGAOOC on Xilinx Virtex 7 690T FPGA. The green line separates in-card computations from out-of-card ones.

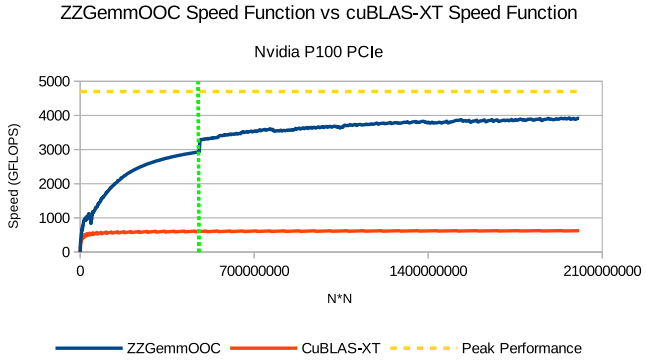

Figure 6.6: Comparison of vendor-optimized library CUBLAS-XT with ZZGemmOOC on Nvidia P100 PCIe GPU. The green line separates in-card computations from out-of-card ones. The dotted yellow line represents the theoretical peak double precision performance of the GPU.

precision floating point performance of Nvidia P100 PCIe is 4.7 TFLOPS. The maximum double-precision floating point performance of ZZGemmOOC is 4 TFLOPS, which constitutes 85% of the peak. ZZGemmOOC is 2 times faster than CUBLAS-XT for small matrix sizes and achieves up to $6x$ speedup for larger ones. The vertical green line in the figure separates the results for in-card matrix multiplication and out-of-card one.

6.4 Summary

HPC users are being enticed to employ the capability of high-performance heterogeneous computing through the provision of hybrid nodes that contain multicore CPUs hosting one or more widely used hardware accelerators such as GPUs, PHIs, and FPGAs.

Hardware acceleration of scientific kernels in HPC poses prominent challenges arising from the limited main memory of accelerators and the tight integration of the accelerators with multicore CPUs via PCI-E communication links. We proposed a library containing interfaces (*HCLOOC*) to cope with the limitations that beset the execution of data parallel applications for large problem sizes on mainstream accelerators. An optimal software pipeline is adopted for communication-computation overlapping. It is designed using the fundamental building blocks, which are OpenCL command queues for FPGAs, Intel offload streams for Intel Xeon Phis, and CUDA streams that allow concurrent utilization of the copy and execution engines provided in Nvidia GPUs.

The library was evaluated using an out-of-card implementation of matrix multiplication of large dense matrices on two hybrid platforms, including a Nvidia K40c GPU, an Intel Xeon Phi 3120P, a Xilinx Virtex 7 690T FPGA and a Nvidia P100 PCIe GPU. We achieved 85% of the peak double-precision floating performance of Nvidia P100 PCIe GPU, and the ZZGemmOOC implementation outperforms the Nvidia's out-of-card matrix multiplication implementation (CUBLAS-XT) by more than $6x$. Our experiments showed 0% performance loss for workloads exceeding the memory capacity of accelerators for GPU, Intel Xeon Phi and Xilinx FPGA.

The software implementations of $HCLOOC$ presented in this chapter can be downloaded from [144], [146] and [163] for Nvidia GPUs, Intel Xeon Phis and FPGAs, respectively.

The library interface design contained creation of a uniform interface for the fundamental building blocks, OpenCL command queues, Intel offload streams and CUDA streams and events which have disparate interfaces. There is no unifying interface, allowing programmers to write reusable out-of-card implementations of their kernels that can run efficiently on different mainstream ac-

celerators. This turned out to be a considerably difficult task, which we aim to present in our future work.

Intel has developed a new library for heterogeneous computing named hStreams [164]. We plan to add support for this new library in *HCLOOC* to replace Intel offloads entirely. We plan to add full support for level-3 BLAS kernels and also triangular and banded matrices in our future work. Furthermore, we plan to provide out-of-card factorizations (LU, QR, Cholesky) that use the out-of-card matrix-matrix multiplication (DGEMM) as a fundamental building block.

We will also look at developing extensions of *HCLOOC* for facilitating programming out-of-card implementations of accelerator kernels for multi-GPU platforms.

Chapter 7

Conclusion

Heterogeneity has become a common and inseparable characteristic of today's high-performance computing (HPC) platforms for achieving not just unprecedented computational power but also to address the well established critical concerns of energy efficiency. Now, the tight interoperation of accelerators, owing to their excellent performance per watt feature, into multicore CPUs has turned into a mainstream attribute of powerful clusters and exascale supercomputers. Nevertheless, heterogeneous computing systems present unprecedented challenges not found in typical homogeneous platforms.

Increasing the level of heterogeneity and the number of processing elements in a single die incurs more severe contention on shared resources, such as LLC, QPI, memory banks, interconnects, PCI-E links and etc., which leads to more complex nodal architectures. In this thesis, we showed that these complexities and NUMA have posed new challenges to the modelling and optimization of hybrid data-parallel applications executing on modern heterogeneous HPC platforms for performance and energy consumption. In Chapters 3 and 4, we experimentally analysed the real-life behaviour of some hybrid applications and illustrated that there is a complex correlation between performance and energy consumption of the applications and problem size. We demonstrated that the speed and dynamic energy profiles of these applications may be non-linear and even non-convex, involving drastic variations, so that these shapes of profiles completely deviate from the characteristics as-

sumed by state-of-the-art algorithms for solving performance and energy optimization problems.

To address these challenges, we modelled the performance and dynamic energy consumption of hybrid applications using a set of application-specific discrete functions of problem size, which are built via direct measurements of performance and dynamic energy consumption. Then, two novel variation-aware data-partitioning algorithms *HPOPTA* and *HEOPTA* were proposed, in Chapters 3 and 4, to solve performance and dynamic energy optimization problems on modern heterogeneous platforms, respectively. The former algorithm generally solves *min-max* INLP single-objective optimization problems with discrete objectives as input. Unlike the state-of-the-art load-balancing approaches, it does not rely on load balancing and often return imbalanced but optimal solutions. *HEOPTA* mainly deals with *min-sum* INLP single-objective optimization problems where inputs are discrete objectives.

In order to obtain trade-off solutions between performance and dynamic energy and also performance and total energy, we have presented two other variation-aware data-partitioning algorithms *HEPOPTA* and *HTPOPTA* in Chapter 5. The Pareto-optimal sets, determined by the algorithms, rarely contain one load-balanced solution whereas the rest are load imbalanced. These algorithms take real-life discrete speed and dynamic energy functions as inputs and consider only one decision variable, which is workload distribution (data partitioning). *HEPOPTA* generally obtains globally Pareto-optimal fronts for any *min-max* and *min-sum* bi-objective optimization problem where inputs are discrete objectives.

In Chapter 6, we also proposed *HCLOOC*, a library allowing out-of-card computation on Nvidia GPUs, Intel Xeon Phis and FPGAs, to address the limitations which prevent computational kernels from running large workloads on these accelerators.

We have published the motivation of this research in [165] and elucidated why new algorithms are required for performance and energy optimization on modern heterogeneous HPC platforms. The results presented in Chapters 3 and 6 have been published in [166] and [167], respectively. We also plan to submit the results of Chapters 4 and 5 to IEEE Transactions on Computers or

IEEE Transactions on Parallel and Distributed Systems.

The potential future work, which could be relevant in the extension of this thesis, includes:

1. Designing parallel versions of the algorithms to reduce their theoretical computations complexities.

2. Developing extensions to the proposed algorithms for optimization of data-parallel applications with 2D and 3D problem dimensions.

3. Considering the cost of inter-node communications for performance and dynamic energy modelling.

4. Analysing our proposed algorithms on heterogeneous platforms involving build-in accelerators on a single chip, such as Accelerator Processing Units (APUs) which include both the CPU and GPU inside a single chip.

5. Studying the efficiency of the algorithms on heterogeneous platforms including non-mainstream accelerators, such as FPGAs, Neural Network Processors (NNPs) and Digital Signal Processors (DSPs).

6. Partitioning the computational cores in an abstract processor into heterogeneous groups and enhancing its performance and energy consumption via workload partitioning.

7. Extensions to *HCLOOC* for supporting hStreams [164], a new library for heterogeneous computing introduced by Intel.

8. Full support for level-2 and level-3 BLAS kernels, with triangular and banded matrices as inputs, and providing out-of-card factorizations (LU, QR, Cholesky) that use the out-of-card matrix-matrix multiplication (DGEMM) as a fundamental building block.

9. Extension of *HCLOOC* for facilitating programming out-of-card implementations of accelerator kernels for multi-GPU platforms.

Bibliography

[1] A. Grama, V. Kumar, A. Gupta, and G. Karypis, *Introduction to parallel computing*. Pearson Education, 2003.

[2] D. Clarke, A. Ilic, A. Lastovetsky, and L. Sousa, "Hierarchical partitioning algorithm for scientific computing on highly heterogeneous CPU+GPU clusters," in *European Conference on Parallel Processing*, pp. 489–501, Springer, 2012.

[3] A. Lastovetsky and R. Reddy, "New model-based methods and algorithms for performance and energy optimization of data parallel applications on homogeneous multicore clusters," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 4, pp. 1119–1133, 2017.

[4] R. R. Schaller, "Moore's law: past, present and future," *IEEE spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[5] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, no. 5, pp. 256–268, 1974.

[6] M. B. Taylor, "Is dark silicon useful? harnessing the four horsemen of the coming dark silicon apocalypse," in *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, pp. 1131–1136, IEEE, 2012.

[7] J. Fruehe, "Multicore processor technology," *Reprinted from Dell Power Solutions www. dell. com/powersolutions (Obtained from the Internet on Mar. 23, 2012)*, pp. 67–72, 2005.

[8] Top500, "Top500." `https://www.top500.org/lists/2018/11/`, 2018.

[9] F. D. Rossi, M. G. Xavier, C. A. De Rose, R. N. Calheiros, and R. Buyya, "E-eco: Performance-aware energy-efficient cloud data center orchestration," *Journal of Network and Computer Applications*, vol. 78, pp. 83–96, 2017.

[10] Y. Kessaci, N. Melab, and E.-G. Talbi, "A pareto-based metaheuristic for scheduling HPC applications on a geographically distributed cloud federation," *Cluster Computing*, vol. 16, no. 3, pp. 451–468, 2013.

[11] M. Cierniak, M. J. Zaki, and W. Li, "Compile-time scheduling algorithms for a heterogeneous network of workstations," *The Computer Journal*, vol. 40, no. 6, pp. 356–372, 1997.

[12] O. Beaumont, V. Boudet, F. Rastello, and Y. Robert, "Matrix multiplication on heterogeneous platforms," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 12, no. 10, pp. 1033–1051, 2001.

[13] A. Kalinov and A. Lastovetsky, "Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers," *J. Parallel Distrib. Comput.*, vol. 61, Apr. 2001.

[14] A. L. Lastovetsky and R. Reddy, "Data partitioning with a realistic performance model of networks of heterogeneous computers," in *Parallel and Distributed Processing Symposium, 2004. Proceedings. 18th International*, p. 104, IEEE, 2004.

[15] A. Lastovetsky and R. Reddy, "Data partitioning with a functional performance model of heterogeneous processors," *International Journal of High Performance Computing Applications*, vol. 21, no. 1, pp. 76–90, 2007.

[16] A. Lastovetsky, L. Szustak, and R. Wyrzykowski, "Model-based optimization of EULAG kernel on Intel Xeon Phi through load imbalancing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 3, pp. 787–797, 2017.

[17] P. K. Smolarkiewicz and W. W. Grabowski, "The multidimensional positive definite advection transport algorithm: Nonoscillatory option," *J. Comput. Phys.*, vol. 86, Feb. 1990.

[18] R. R. Manumachu and A. Lastovetsky, "Parallel data partitioning algorithms for optimization of data-parallel applications on modern extreme-scale multicore platforms for performance and energy," *IEEE Access*, vol. 6, pp. 69075–69106, 2018.

[19] OpenBLAS, "OpenBLAS: An optimized BLAS library." `http://www.openblas.net/`, 2016.

[20] FFTW, "FFTW: A fast, free c FFT library." `http://www.fftw.org/`, 2016.

[21] A. Lastovetsky and R. Reddy, "Data distribution for dense factorization on computers with memory heterogeneity," *Parallel Computing*, vol. 33, Dec. 2007.

[22] A. Ilić, F. Pratas, P. Trancoso, and L. Sousa, "High-performance computing on heterogeneous systems: Database queries on CPU and GPU," *High Performance Scientific Computing with Special Emphasis on Current Capabilities and Future Perspectives*, pp. 202–222, 2010.

[23] D. Clarke, A. Lastovetsky, and V. Rychkov, "Dynamic load balancing of parallel computational iterative routines on highly heterogeneous HPC platforms," *Parallel Processing Letters*, vol. 21, no. 02, pp. 195–217, 2011.

[24] D. Clarke, A. L. Lastovetsky, and V. Rychkov, "Column-based matrix partitioning for parallel matrix multiplication on heterogeneous processors based on functional performance models," in *Euro-Par 2011: Parallel Processing Workshops*, vol. 7155 of *Lecture Notes in Computer Science*, Springer-Verlag, 2012.

[25] X. Liu, Z. Zhong, and K. Xu, "A hybrid solution method for CFD applications on GPU-accelerated hybrid HPC platforms," *Future Generation Computer Systems*, vol. 56, pp. 759–765, 2016.

[26] M. Radmanović, D. Gajić, and R. Stanković, "Efficient computation of galois field expressions on hybrid CPU-GPU platforms.," *Journal of Multiple-Valued Logic & Soft Computing*, vol. 26, 2016.

[27] A. Ilic and L. Sousa, "Simultaneous multi-level divisible load balancing for heterogeneous desktop systems," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*, pp. 683–690, IEEE, 2012.

[28] J. Colaço, A. Matoga, A. Ilic, N. Roma, P. Tomás, and R. Chaves, "Transparent application acceleration by intelligent scheduling of shared library calls on heterogeneous systems," in *Parallel Processing and Applied Mathematics*, pp. 693–703, Springer, 2013.

[29] V. Cardellini, A. Fanfarillo, and S. Filippone, "Heterogeneous sparse matrix computations on hybrid GPU/CPU platforms.," in *PARCO*, pp. 203–212, 2013.

[30] C.-Y. Yang, J.-J. Chen, T.-W. Kuo, and L. Thiele, "An approximation scheme for energy-efficient scheduling of real-time tasks in heterogeneous multiprocessor systems," in *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pp. 694–699, IEEE, 2009.

[31] J. Li and J. F. Martínez, "Power-performance considerations of parallel computing on chip multiprocessors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 2, no. 4, pp. 397–422, 2005.

[32] X. Fan, W.-D. Weber, and L. A. Barroso, "Power provisioning for a warehouse-sized computer," in *ACM SIGARCH computer architecture news*, vol. 35, pp. 13–23, ACM, 2007.

[33] K. Meng, R. Joseph, R. P. Dick, and L. Shang, "Multi-optimization power management for chip multiprocessors," in *Proceedings of the 17th in-*

ternational conference on Parallel architectures and compilation techniques, pp. 177–186, ACM, 2008.

[34] O. Azizi, A. Mahesri, B. C. Lee, S. J. Patel, and M. Horowitz, "Energy-performance tradeoffs in processor architecture and circuit design: a marginal cost analysis," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 26–36, 2010.

[35] J. Zhao, G. Sun, G. H. Loh, and Y. Xie, "Optimizing GPU energy efficiency with 3D die-stacking graphics memory and reconfigurable memory interface," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 10, no. 4, p. 24, 2013.

[36] A. W. Lewis, S. Ghosh, and N.-F. Tzeng, "Run-time energy consumption estimation based on workload in server systems.," *HotPower*, vol. 8, pp. 17–21, 2008.

[37] C. Lively, X. Wu, V. Taylor, S. Moore, H.-C. Chang, C.-Y. Su, and K. Cameron, "Power-aware predictive models of hybrid (mpi/openmp) scientific applications on multicore systems," *Computer Science-Research and Development*, vol. 27, no. 4, pp. 245–253, 2012.

[38] M. Rofouei, T. Stathopoulos, S. Ryffel, W. Kaiser, and M. Sarrafzadeh, "Energy-aware high performance computing with graphic processing units," in *Workshop on power aware computing and system*, 2008.

[39] S. Hong and H. Kim, "An integrated GPU power and performance model," in *ACM SIGARCH Computer Architecture News*, vol. 38, pp. 280–289, ACM, 2010.

[40] H. Nagasaka, N. Maruyama, A. Nukada, T. Endo, and S. Matsuoka, "Statistical power modeling of GPU kernels using performance counters," in *Green Computing Conference, 2010 International*, pp. 115–122, IEEE, 2010.

[41] W. L. Bircher and L. K. John, "Complete system power estimation using processor performance events," *IEEE Transactions on Computers*, vol. 61, no. 4, pp. 563–577, 2012.

[42] K. Kasichayanula, D. Terpstra, P. Luszczek, S. Tomov, S. Moore, and G. D. Peterson, "Power aware computing on GPUs," in *Application Accelerators in High Performance Computing (SAAHPC), 2012 Symposium on*, pp. 64–73, IEEE, 2012.

[43] J. W. Choi, D. Bedard, R. Fowler, and R. Vuduc, "A roofline model of energy," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 661–672, IEEE, 2013.

[44] Y. S. Shao and D. Brooks, "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor," in *Low Power Electronics and Design (ISLPED), 2013 IEEE International Symposium on*, pp. 389–394, IEEE, 2013.

[45] R. R. Manumachu and A. Lastovetsky, "Bi-objective optimization of data-parallel applications on homogeneous multicore clusters for performance and energy," *IEEE Transactions on Computers*, vol. 67, no. 2, pp. 160–177, 2018.

[46] R. Reddy Manumachu and A. L. Lastovetsky, "Design of self-adaptable data parallel applications on multicore clusters automatically optimized for performance and energy through load distribution," *Concurrency and Computation: Practice and Experience*, vol. 0, no. 0, p. e4958.

[47] J. Kołodziej, S. U. Khan, L. Wang, and A. Y. Zomaya, "Energy efficient genetic-based schedulers in computational grids," *Concurrency and Computation: Practice and Experience*, vol. 27, no. 4, pp. 809–829, 2015.

[48] I. Ahmad, S. Ranka, and S. U. Khan, "Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–6, IEEE, 2008.

[49] B. Subramaniam and W.-c. Feng, "Statistical power and performance modeling for optimizing the energy efficiency of scientific computing," in *Proceedings of the 2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*, pp. 139–146, IEEE Computer Society, 2010.

[50] Y. Inadomi, T. Patki, K. Inoue, M. Aoyagi, B. Rountree, M. Schulz, D. Lowenthal, Y. Wada, K. Fukazawa, M. Ueda, *et al.*, "Analyzing and mitigating the impact of manufacturing variability in power-constrained supercomputing," in *High Performance Computing, Networking, Storage and Analysis, 2015 SC-International Conference for*, pp. 1–12, IEEE, 2015.

[51] N. Gholkar, F. Mueller, and B. Rountree, "Power tuning HPC jobs on power-constrained systems," in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, pp. 179–191, ACM, 2016.

[52] K. O'Brien, I. Pietri, R. Reddy, A. Lastovetsky, and R. Sakellariou, "A survey of power and energy predictive models in HPC systems and applications," *ACM Computing Surveys (CSUR)*, vol. 50, no. 3, p. 37, 2017.

[53] J. C. McCullough, Y. Agarwal, J. Chandrashekar, S. Kuppuswamy, A. C. Snoeren, and R. K. Gupta, "Evaluating the effectiveness of model-based power characterization," in *USENIX Annual Technical Conf*, vol. 20, 2011.

[54] D. Hackenberg, T. Ilsche, R. Schöne, D. Molka, M. Schmidt, and W. E. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," in *Performance analysis of systems and software (ISPASS), 2013 IEEE international symposium on*, pp. 194–204, IEEE, 2013.

[55] A. Shahid, M. Fahad, R. Reddy, and A. Lastovetsky, "Additivity: A selection criterion for performance events for reliable energy predictive model-

ing," *Supercomputing Frontiers and Innovations*, vol. 4, no. 4, pp. 50–65, 2017.

[56] S. F. Piraghaj, A. V. Dastjerdi, R. N. Calheiros, and R. Buyya, "A survey and taxonomy of energy efficient resource management techniques in platform as a service cloud," in *Handbook of Research on End-to-End Cloud Computing Architecture Design*, pp. 410–454, IGI Global, 2017.

[57] Y. Liu, H. Zhu, K. Lu, and X. Wang, "Self-adaptive management of the sleep depths of idle nodes in large scale systems to balance between energy consumption and response times," in *Cloud Computing Technology and Science (CloudCom), 2012 IEEE 4th International Conference on*, pp. 633–639, IEEE, 2012.

[58] A. Benoit, L. Lefèvre, A.-C. Orgerie, and I. Raïs, "Reducing the energy consumption of large-scale computing systems through combined shutdown policies with multiple constraints," *The International Journal of High Performance Computing Applications*, vol. 32, no. 1, pp. 176–188, 2018.

[59] K. Chen, J. Lenhardt, and W. Schiffmann, "Improving energy efficiency of web servers by using a load distribution algorithm and shutting down idle nodes," in *Cluster, Cloud and Grid Computing (CCGrid), 2015 15th IEEE/ACM International Symposium on*, pp. 745–748, IEEE, 2015.

[60] K. Rajamani and C. Lefurgy, "On evaluating request-distribution schemes for saving energy in server clusters," in *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, pp. 111–122, IEEE, 2003.

[61] M. Mezmaz, N. Melab, Y. Kessaci, Y. C. Lee, E.-G. Talbi, A. Y. Zomaya, and D. Tuyttens, "A parallel bi-objective hybrid metaheuristic for energy-aware scheduling for cloud computing systems," *Journal of Parallel and Distributed Computing*, vol. 71, no. 11, pp. 1497–1508, 2011.

[62] H. M. Fard, R. Prodan, J. J. D. Barrionuevo, and T. Fahringer, "A multi-objective approach for workflow scheduling in heterogeneous environments," in *Cluster, Cloud and Grid Computing (CCGrid), 2012 12th IEEE/ACM International Symposium on*, pp. 300–309, IEEE, 2012.

[63] A. Beloglazov, J. Abawajy, and R. Buyya, "Energy-aware resource allocation heuristics for efficient management of data centers for cloud computing," *Future generation computer systems*, vol. 28, no. 5, pp. 755–768, 2012.

[64] J. J. Durillo, V. Nae, and R. Prodan, "Multi-objective energy-efficient workflow scheduling using list-based heuristics," *Future Generation Computer Systems*, vol. 36, pp. 221–236, 2014.

[65] V. W. Freeh, D. K. Lowenthal, F. Pan, N. Kappiah, R. Springer, B. L. Rountree, and M. E. Femal, "Analyzing the energy-time trade-off in high-performance computing applications," *IEEE Transactions on Parallel & Distributed Systems*, no. 6, pp. 835–848, 2007.

[66] P. Balaprakash, A. Tiwari, and S. M. Wild, "Multi objective optimization of HPC kernels for performance, power, and energy," in *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pp. 239–260, Springer, 2013.

[67] M. A. Aba, L. Zaourar, and A. Munier, "Approximation algorithm for scheduling a chain of tasks on heterogeneous systems," in *European Conference on Parallel Processing*, pp. 353–365, Springer, 2017.

[68] S. Song, C.-Y. Su, R. Ge, A. Vishnu, and K. W. Cameron, "Iso-energy-efficiency: An approach to power-constrained parallel computation," in *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pp. 128–139, IEEE, 2011.

[69] J. Demmel, A. Gearhart, B. Lipshitz, and O. Schwartz, "Perfect strong scaling using no additional energy," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 649–660, IEEE, 2013.

[70] J. M. Marszałkowski, M. Drozdowski, and J. Marszałkowski, "Time and energy performance of parallel systems with hierarchical memory," *Journal of Grid Computing*, vol. 14, no. 1, pp. 153–170, 2016.

[71] K. M. Tarplee, R. Friese, A. A. Maciejewski, H. J. Siegel, and E. K. Chong, "Energy and makespan tradeoffs in heterogeneous computing systems using efficient linear programming techniques," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 6, pp. 1633–1646, 2016.

[72] E. Gabaldon, J. L. Lerida, F. Guirado, and J. Planes, "Blacklist mutiobjective genetic algorithm for energy saving in heterogeneous environments," *The Journal of Supercomputing*, vol. 73, no. 1, pp. 354–369, 2017.

[73] A. Chakrabarti, S. Parthasarathy, and C. Stewart, "A pareto framework for data analytics on heterogeneous systems: Implications for green energy usage and performance," in *Parallel Processing (ICPP), 2017 46th International Conference on*, pp. 533–542, IEEE, 2017.

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

[75] Nvidia, "CUDA zone." `https://developer.nvidia.com/cuda-zone`, 2018.

[76] A. T. Chronopoulos, D. Grosu, A. M. Wissink, M. Benche, and J. Liu, "An efficient 3D grid based scheduling for heterogeneous systems," *Journal of Parallel and Distributed Computing*, vol. 63, no. 9, pp. 827 – 837, 2003. Special Section on the Best Papers from the 2002 International Parallel and Distributed Processing Symposium.

[77] Y. Ogata, T. Endo, N. Maruyama, and S. Matsuoka, "An efficient, model-based CPU-GPU heterogeneous FFT library," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–10, IEEE, 2008.

[78] C. Yang, F. Wang, Y. Du, J. Chen, J. Liu, H. Yi, and K. Lu, "Adaptive optimization for petascale heterogeneous CPU/GPU computing," in *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pp. 19–28, IEEE, 2010.

[79] H. Khaleghzadeh, H. Deldari, R. Reddy, and A. Lastovetsky, "Hierarchical multicore thread mapping via estimation of remote communication," *The Journal of Supercomputing*, vol. 74, no. 3, pp. 1321–1340, 2018.

[80] M. D. Linderman, J. D. Collins, H. Wang, and T. H. Meng, "Merge: a programming model for heterogeneous multi-core systems," in *ACM SIGOPS operating systems review*, vol. 42, pp. 287–296, ACM, 2008.

[81] G. Quintana-Ortí, F. D. Igual, E. S. Quintana-Ortí, and R. A. van de Geijn, "Solving dense linear systems on platforms with multiple hardware accelerators," *SIGPLAN Not.*, vol. 44, pp. 121–130, Feb. 2009.

[82] C. Augonnet, S. Thibault, and R. Namyst, "Automatic calibration of performance models on heterogeneous multicore architectures," in *3rd Workshop on Highly Parallel Processing on a Chip (HPPC 2009)*, Aug. 2009.

[83] F. Song, S. Tomov, and J. Dongarra, "Enabling and scaling matrix computations on heterogeneous multi-core and multi-GPU systems," in *Proceedings of the 26th ACM international conference on Supercomputing*, pp. 365–376, ACM, 2012.

[84] K. Kyriakopoulos, A. T. Chronopoulos, and L. Ni, "An optimal scheduling scheme for tiling in distributed systems," in *Cluster Computing, 2007 IEEE International Conference on*, pp. 267–274, IEEE, 2007.

[85] G. Liu, J. Park, and D. Marculescu, "Dynamic thread mapping for high-performance, power-efficient heterogeneous many-core systems," in *Computer Design (ICCD), 2013 IEEE 31st International Conference on*, pp. 54–61, IEEE, 2013.

[86] K. Schloegel, G. Karypis, and V. Kumar, "A unified algorithm for load-balancing adaptive scientific simulations," in *Supercomputing, ACM/IEEE 2000 Conference*, pp. 59–59, Nov 2000.

[87] U. V. Catalyurek, E. G. Boman, K. D. Devine, D. Bozdag, R. Heaphy, and L. A. Riesen, "Hypergraph-based dynamic load balancing for adaptive scientific computations," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, pp. 1–11, IEEE, 2007.

[88] G. Cybenko, "Dynamic load balancing for distributed memory multiprocessors," *J. Parallel Distrib. Comput.*, vol. 7, pp. 279–301, Oct. 1989.

[89] J. M. Bahi, S. Contassot-Vivier, and R. Couturier, "Dynamic load balancing and efficient load estimators for asynchronous iterative algorithms," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 16, no. 4, pp. 289–299, 2005.

[90] A. Legrand, H. Renard, Y. Robert, and F. Vivien, "Mapping and load-balancing iterative computations," *IEEE Trans. Parallel Distrib. Syst.*, vol. 15, June 2004.

[91] R. L. Cariño and I. Banicescu, "Dynamic load balancing with adaptive factoring methods in scientific applications," *The Journal of Supercomputing*, vol. 44, no. 1, pp. 41–63, 2008.

[92] J. A. Martínez, E. M. Garzón, A. Plaza, and I. García, "Automatic tuning of iterative computation on heterogeneous multiprocessors with ADITHE," *J. Supercomput.*, vol. 58, Nov. 2011.

[93] P. Joseph, K. Vaswani, and M. J. Thazhuthaveetil, "A predictive performance model for superscalar processors," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 161–170, IEEE Computer Society, 2006.

[94] E. Grobelny, D. Bueno, I. Troxel, A. D. George, and J. S. Vetter, "FASE: A framework for scalable performance prediction of HPC systems and applications," *Simulation*, vol. 83, no. 10, pp. 721–745, 2007.

[95] S. Williams, A. Waterman, and D. Patterson, "Roofline: an insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, 2009.

[96] K.-H. Kim, K. Kim, and Q.-H. Park, "Performance analysis and optimization of three-dimensional FDTD on GPU using roofline model," *Computer Physics Communications*, vol. 182, no. 6, pp. 1201–1207, 2011.

[97] C. Nugteren and H. Corporaal, "The boat hull model: adapting the roofline model to enable performance prediction for parallel computing," in *ACM Sigplan Notices*, vol. 47, pp. 291–292, ACM, 2012.

[98] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An adaptive performance modeling tool for GPU architectures," in *ACM Sigplan Notices*, vol. 45, pp. 105–114, ACM, 2010.

[99] Y. Zhang and J. D. Owens, "A quantitative performance analysis model for GPU architectures," in *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on*, pp. 382–393, IEEE, 2011.

[100] M. R. Meswani, L. Carrington, D. Unat, A. Snavely, S. Baden, and S. Poole, "Modeling and predicting performance of high performance computing applications on hardware accelerators," *The International Journal of High Performance Computing Applications*, vol. 27, no. 2, pp. 89–108, 2013.

[101] J. Shen, A. L. Varbanescu, Y. Lu, P. Zou, and H. Sips, "Workload partitioning for accelerating applications on heterogeneous platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 9, pp. 2766–2780, 2016.

[102] F. Zhang, B. Wu, J. Zhai, B. He, and W. Chen, "FinePar: Irregularity-aware fine-grained workload partitioning on integrated architectures," in *Proceedings of the 2017 International Symposium on Code Generation and Optimization*, pp. 27–38, IEEE Press, 2017.

[103] C. Rosales, A. Gómez-Iglesias, S. Liu, F. Chen, L. Huang, H. Liu, A. Lamas-Linares, and J. Cazes, "Performance prediction of HPC applications on Intel processors," in *Parallel and Distributed Processing Symposium Workshops (IPDPSW), 2017 IEEE International*, pp. 1325–1332, IEEE, 2017.

[104] M. A. Obaida, J. Liu, G. Chennupati, N. Santhi, and S. Eidenbenz, "Parallel application performance prediction using analysis based models and HPC simulations," in *Proceedings of the 2018 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation*, pp. 49–59, ACM, 2018.

[105] N. Ding, S. Xu, Z. Song, B. Zhang, J. Li, and Z. Zheng, "Using hardware counter-based performance model to diagnose scaling issues of HPC applications," *Neural Computing and Applications*, pp. 1–13, 2018.

[106] A. Lastovetsky and R. Reddy, "A novel algorithm of optimal matrix partitioning for parallel dense factorization on heterogeneous processors," in *International Conference on Parallel Computing Technologies*, pp. 261–275, Springer, 2007.

[107] O. Beaumont, V. Boudet, A. Petitet, F. Rastello, and Y. Robert, "A proposal for a heterogeneous cluster ScaLAPACK (dense linear solvers)," *IEEE Transactions on Computers*, vol. 50, no. 10, pp. 1052–1070, 2001.

[108] M. Fatica, "Accelerating linpack with CUDA on heterogenous clusters," in *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pp. 46–51, ACM, 2009.

[109] R. Wyrzykowski, L. Szustak, K. Rojek, and A. Tomas, "Towards efficient decomposition and parallelization of mpdata on hybrid CPU-GPU cluster," in *International Conference on Large-Scale Scientific Computing*, pp. 457–464, Springer, 2013.

[110] A. Lastovetsky and R. Reddy, "Data partitioning for multiprocessors with memory heterogeneity and memory constraints," *Scientific Programming*, vol. 13, no. 2, pp. 93–112, 2005.

[111] W. Zhang, X. Ji, B. Song, S. Yu, H. Chen, T. Li, P. C. Yew, and W. Zhao, "VarCatcher: A framework for tackling performance variability of parallel workloads on multi-core," *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, pp. 1215–1228, April 2017.

[112] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye, "Energy-driven integrated hardware-software optimizations using SimplePower," *ACM SIGARCH Computer Architecture News*, vol. 28, no. 2, pp. 95–106, 2000.

[113] D. Brooks, V. Tiwari, and M. Martonosi, *Wattch: A framework for architectural-level power analysis and optimizations*, vol. 28. ACM, 2000.

[114] D. Brooks, M. Martonosi, J.-D. Wellman, and P. Bose, "Power-performance modeling and tradeoff analysis for a high end microprocessor," in *International Workshop on Power-Aware Computer Systems*, pp. 126–136, Springer, 2000.

[115] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 469–480, ACM, 2009.

[116] J. Lim, N. B. Lakshminarayana, H. Kim, W. Song, S. Yalamanchili, and W. Sung, "Power modeling for GPU architectures using McPAT," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 19, no. 3, p. 26, 2014.

[117] J. Chen, B. Li, Y. Zhang, L. Peng, and J.-k. Peir, "Statistical GPU power analysis using tree-based methods," in *Green Computing Conference and Workshops (IGCC), 2011 International*, pp. 1–6, IEEE, 2011.

[118] J. Lucas, S. Lal, M. Andersch, M. Alvarez-Mesa, and B. Juurlink, "How a single chip causes massive power bills GPUSimPow: A GPGPU power

simulator," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pp. 97–106, IEEE, 2013.

[119] S. Song, C. Su, B. Rountree, and K. W. Cameron, "A simplified and accurate model of power-performance efficiency on emergent GPU architectures," in *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pp. 673–686, IEEE, 2013.

[120] G. Kestor, R. Gioiosa, D. J. Kerbyson, and A. Hoisie, "Enabling accurate power profiling of HPC applications on exascale systems," in *Proceedings of the 3rd International Workshop on Runtime and Operating Systems for Supercomputers*, p. 4, ACM, 2013.

[121] M. Jarus, A. Oleksiak, T. Piontek, and J. Węglarz, "Runtime power usage estimation of HPC servers for various classes of real-life applications," *Future Generation Computer Systems*, vol. 36, pp. 299–310, 2014.

[122] Z. Al-Khatib and S. Abdi, "Operand-value-based modeling of dynamic energy consumption of soft processors in FPGA," in *International Symposium on Applied Reconfigurable Computing*, pp. 65–76, Springer, 2015.

[123] S. Kamil, J. Shalf, and E. Strohmaier, "Power efficiency in high performance computing," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pp. 1–8, IEEE, 2008.

[124] S. Rivoire, P. Ranganathan, and C. Kozyrakis, "A comparison of high-level full-system power models.," *HotPower*, vol. 8, no. 2, pp. 32–39, 2008.

[125] J. Choi, M. Dukhan, X. Liu, and R. Vuduc, "Algorithmic time, energy, and power on candidate HPC compute building blocks," in *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 447–457, IEEE, 2014.

[126] L. Gu, J. Siegel, and X. Li, "Using GPUs to compute large out-of-card FFTs," in *Proceedings of the International Conference on Supercomputing*, ICS '11, pp. 255–264, ACM, 2011.

[127] X. Mu, H.-X. Zhou, K. Chen, and W. Hong, "Higher order method of moments with a parallel out-of-core LU solver on GPU/CPU platform," *IEEE Transactions on Antennas and Propagation*, vol. 62, no. 11, pp. 5634–5646, 2014.

[128] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on heterogeneous multicore and Multi-GPU systems using functional performance models of Data-Parallel applications," in *2012 IEEE International Conference on Cluster Computing (Cluster 2012)*, pp. 191–199, 24-28 September 2012.

[129] Z. Zhong, *Optimization of Data-Parallel Scientific Applications on Highly Heterogeneous Modern HPC Platforms*. PhD thesis, University College Dublin, 2014.

[130] J. Wu and J. Jaja, "Achieving native GPU performance for out-of-card large dense matrix multiplication," *Parallel Processing Letters*, vol. 26, no. 02, p. 1650007, 2016.

[131] A. Sabne, P. Sakdhnagool, and R. Eigenmann, "Scaling large-data computations on multi-GPU accelerators," in *Proceedings of the 27th international ACM conference on International conference on supercomputing*, pp. 443–454, ACM, 2013.

[132] K. Shirahata, H. Sato, and S. Matsuoka, "Out-of-core GPU memory management for mapreduce-based large-scale graph processing," in *Cluster Computing (CLUSTER), 2014 IEEE International Conference on*, pp. 221–229, IEEE, 2014.

[133] K. Kabir, A. Haidar, S. Tomov, A. Bouteiller, and J. Dongarra, "A framework for out of memory SVD algorithms," in *International Supercomputing Conference*, pp. 158–178, Springer, 2017.

[134] A. Haidar, K. Kabir, D. Fayad, S. Tomov, and J. Dongarra, "Out of memory SVD solver for big data," in *High Performance Extreme Computing Conference (HPEC), 2017 IEEE*, pp. 1–7, IEEE, 2017.

[135] I. Yamazaki, S. Tomov, and J. Dongarra, "Non-GPU-resident symmetric indefinite factorization," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 5, p. e4012, 2017.

[136] CUBLAS-XT, "CUBLAS-XT: Multi-GPU version of CUBLAS library supporting out-of-core routines." `https://developer.nvidia.com/cublas`, 2016.

[137] NVIDIA, "CUDA C Programming Guide." `https://docs.nvidia.com/cuda/cuda-c-programming-guide/`, 2016.

[138] R. Edgar, "SciGPU-GEMM." `https://github.com/YaohuiZeng/scigpugemm`, 2009.

[139] D. Martin, "High performance computing linpack benchmark for CUDA." `https://github.com/avidday/hpl-cuda`, 2010.

[140] S. Tomov, J. Dongarra, V. Volkov, and J. Demmel, "Magma library," *Univ. of Tennessee and Univ. of California, Knoxville, TN, and Berkeley, CA*, 2009.

[141] S. Tomov, J. Dongarra, and M. Baboulin, "Towards dense linear algebra for hybrid GPU accelerated manycore systems," *Parallel Computing*, vol. 36, no. 5-6, pp. 232–240, 2010.

[142] Z. Zhong, V. Rychkov, and A. Lastovetsky, "Data partitioning on multicore and multi-GPU platforms using functional performance models," *Computers, IEEE Transactions on*, vol. 64, no. 9, pp. 2506–2518, 2015.

[143] I. Corporation, "Intel Math Kernel Library-Intel MKL BLAS." `https://software.intel.com/en-us/mkl/features/linear-algebra`, 2018.

[144] H. Khaleghzadeh, Z. Zhong, R. Reddy, and A. Lastovetsky, "ZZGEMMOOC: A package for out-of-card DGEMM on GPU." `https://git.ucd.ie/hcl/zzgemmooc.git`, 2017.

[145] Nvidia, "CUDA toolkit documentation." `http://docs.nvidia.com/cuda/cublas/index.html#axzz4kRVc2o6B`, 2017.

[146] H. Khaleghzadeh, Z. Zhong, R. Reddy, and A. Lastovetsky, "XeonPhiOOC: A package for out-of-card DGEMM on Xeon Phi." `https://git.ucd.ie/manumachu/xeonphiooc.git`, 2017.

[147] I. Corporation, "Intel Math Kernel Library-Intel MKL FFT." `https://software.intel.com/en-us/mkl/features/fft`, 2018.

[148] Nvidia, "Optimized FFT routines for Nvidia graphics processors." `https://docs.nvidia.com/cuda/cufft/index.html`, 2018.

[149] A. Lastovetsky, R. Reddy, and R. Higgins, "Building the functional performance model of a processor," in *Proceedings of the 2006 ACM symposium on Applied computing*, pp. 746–753, ACM, 2006.

[150] H. Khaleghzadeh, R. Reddy, and A. Lastovetsky, "HPOPTA: Heterogeneous model-based data partitioning algorithm for optimization of data-parallel applications for performance." `https://git.ucd.ie/hkhaleghzadeh/hpopt.git`, 2017.

[151] J. Hsu, "Three paths to exascale supercomputing," *IEEE Spectrum*, vol. 53, no. 1, pp. 14–15, 2016.

[152] DOE, "Preliminary conceptual design for an exascale computing initiative." `https://science.energy.gov/~/media/ascr/ascac/pdf/meetings/20141121/Exascale_Preliminary_Plan_V11_sb03c.pdf`, 2014.

[153] D. Economou, S. Rivoire, C. Kozyrakis, and P. Ranganathan, "Full-system power analysis and modeling for server environments," International Symposium on Computer Architecture-IEEE, 2006.

[154] B. Goel, S. A. McKee, R. Gioiosa, K. Singh, M. Bhadauria, and M. Cesati, "Portable, scalable, per-core power estimation for intelligent resource management," in *International Conference on Green Computing*, pp. 135–146, IEEE, 2010.

[155] R. Basmadjian, N. Ali, F. Niedermeier, H. De Meer, and G. Giuliani, "A methodology to predict the power consumption of servers in data centres," in *Proceedings of the 2nd international conference on energy-efficient computing and networking*, pp. 1–10, ACM, 2011.

[156] HCL, "HCLWattsUp: API for power and energy measurements using WattsUp Pro Meter." `http://git.ucd.ie/hcl/hclwattsup`, 2016.

[157] H. Khaleghzadeh, R. Reddy, and A. Lastovetsky, "HEOPTA: Heterogeneous model-based data partitioning algorithm for optimization of data-parallel applications for dynamic energy." `https://git.ucd.ie/hkhaleghzadeh/heopt`, 2018.

[158] H. Khaleghzadeh, R. Reddy, and A. Lastovetsky, "HEPOPTA: Heterogeneous model-based data partitioning algorithm for optimization of data-parallel applications for dynamic energy and performance and for total energy and performance." `https://git.ucd.ie/hkhaleghzadeh/hepopt`, 2018.

[159] J. Suzuki, Y. Hayashi, M. Kan, S. Miyakawa, T. Takenaka, T. Araki, and M. Kitsuregawa, "Victream: Computing framework for out-of-core processing on multiple GPUs," in *Proceedings of the Fourth IEEE/ACM International Conference on Big Data Computing, Applications and Technologies*, pp. 179–188, ACM, 2017.

[160] Khronos OpenCL Registry, "OpenCL Command Queues." `https://www.khronos.org/registry/OpenCL/specs/opencl-2.2.pdf`, 2017.

[161] Intel, "Programming for Intel MIC architecture." `https://software.intel.com/en-us/node/684368`, 2017.

[162] NVIDIA, "Tesla K40 GPU accelerator." `http://www.nvidia.com/content/PDF/kepler/Tesla-K40-PCIe-Passive-Board-Spec-BD-06902-001_v05.pdf`, 2013.

[163] H. Khaleghzadeh, Z. Zhong, R. Reddy, and A. Lastovetsky., "FPGAOOC: A package for out-of-card DGEMM on FPGA." `https://git.ucd.ie/hcl/fpgagemm.git`, 2017.

[164] C. J. Newburn, G. Bansal, M. Wood, L. Crivelli, J. Planas, A. Duran, P. Souza, L. Borges, P. Luszczek, S. Tomov, *et al.*, "Heterogeneous streaming," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 611–620, IEEE, 2016.

[165] A. Lastovetsky, M. Fahad, H. Khaleghzadeh, S. Khokhriakov, R. Reddy, A. Shahid, L. Szustak, and R. Wyrzykowski, "How pre-multicore methods and algorithms perform in multicore era," in *International Conference on High Performance Computing*, pp. 527–539, Springer, 2018.

[166] H. Khaleghzadeh, R. R. Manumachu, and A. Lastovetsky, "A novel data-partitioning algorithm for performance optimization of data-parallel applications on heterogeneous HPC platforms," *IEEE Transactions on Parallel and Distributed Systems*, vol. 29, no. 10, pp. 2176–2190, 2018.

[167] H. Khaleghzadeh, Z. Zhong, R. Reddy, and A. Lastovetsky, "Out-of-core implementation for accelerator kernels on heterogeneous clouds," *The Journal of Supercomputing*, vol. 74, no. 2, pp. 551–568, 2018.

Appendix A

Experimental Methodology

To make sure the experimental results are reliable, we follow the methodology described below:

- The server is fully reserved and dedicated to the experiments during execution. We also ensure that there are no drastic fluctuations in the load due to abnormal events in the server by monitoring its load continuously for a week using the tool *sar*. Insignificant variation in the load was observed during this monitoring period suggesting normal and clean behaviour of the server.
- A given hybrid application is executed simultaneously on all abstract processors to obtain a data point in its speed or dynamic energy functions. The application is repeatedly executed until the sample mean of measurements lies in a user-defined confidence interval and a user-defined precision is achieved. We set the confidence interval as 95% and the precision as 0.1 (10%) for our experiments. For this purpose, Student's t-test is used assuming that the individual observations are independent and their population follows the normal distribution. We verify the validity of these assumptions by plotting the distributions of observations.
- We set *OMP_PLACES* and *OMP_PROC_BIND* environment variables to bind all the threads of a hybrid application to CPU cores.

A.0.1 Methodology to Measure Execution Time and Energy Consumption

Suppose there exists a hybrid application, which is named *app*, consisting of three sample kernels, *Kernel_cpu*, *Kernel_gpu* and *Kernel_phi*, which run in parallel. The goal is to measure the execution time and the dynamic energy consumption of kernels in the application. To do this, we instrument the sample application as shown in Algorithm 10. This instrumented application returns the execution time of each kernel and the energy consumption of all the three kernels.

Algorithm 10 Instrumentation of a sample application (*app*) consisting of three kernels, running on CPU, GPU and PHI simultaneously.

1: HCL_WATTSUP_START( )
2: #pragma parallel
3: **Begin**
4: $te_{cpu}1 \leftarrow gettimeofday()$
5: KERNEL_CPU( )
6: $te_{cpu}2 \leftarrow gettimeofday()$
7: **End**
8: **Begin**
9: $te_{gpu}1 \leftarrow gettimeofday()$
10: KERNEL_GPU( )
11: $te_{gpu}2 \leftarrow gettimeofday()$
12: **End**
13: **Begin**
14: $te_{phi}1 \leftarrow gettimeofday()$
15: KERNEL_PHI( )
16: $te_{phi}2 \leftarrow gettimeofday()$
17: **End**
18: $energy_{app} \leftarrow$ HCL_WATTSUP_STOP( )
19: $te_{cpu} \leftarrow te_{cpu}2 - te_{cpu}1$
20: $te_{gpu} \leftarrow te_{gpu}2 - te_{gpu}1$
21: $te_{phi} \leftarrow te_{phi}2 - te_{phi}1$
22: **return** $(te_{cpu}, te_{gpu}, te_{phi}, energy_{app})$

Methodology to Measure Execution Time

We instrument each kernel in the hybrid application (*app*) by using the member function *gettimeofday()* of the Linux library *sys/time.h* to measure its execution time separately. As shown in Algorithm 10, the execution times are stored in variables te_{cpu}, te_{gpu} and te_{phi} and are returned at the end of the application execution.

Methodology to Measure the Energy Consumption

We have two heterogeneous hybrid nodes. Each node is facilitated with one WattsUp Pro power meter that sits between the wall A/C outlets and the input power sockets of the node. These power meters capture the total power consumption of the node. The power meters have data cables connected to one USB port of the node. One Perl script collects the data from the power meter using the serial USB interface. The execution of these scripts is non-intrusive and consumes insignificant power.

The power meters are periodically calibrated using an ANSI C12.20 revenue-grade power meter, Yokogawa WT210. The maximum sampling speed of the power meters is one sample every second. The accuracy specified in the data-sheets is $\pm 3\%$. The minimum measurable power is 0.5 watts, and the accuracy at 0.5 watts is ± 0.3 watts.

We use *HCLWattsUp* API, which gathers the readings from the power meters to determine the average power and energy consumption during the execution of an application for the whole node. *HCLWattsUp* API [156] also provides two macros: *HCL_WATTSUP_START* and *HCL_WATTSUP_STOP*. The *HCL_WATTSUP_START* macro starts gathering power readings from the power meter using the aforementioned Perl script, whereas the *HCL_WATTSUP_STOP* stops gathering and return the total energy as a sum of these power readings.

To measure the amount of dynamic energy consumed by the application, we invoke *HCL_WATTSUP_START* (Line 1) and *HCL_WATTSUP_STOP* (Line 18) macros as shown in Algorithm 10. The consumed energy is stored in the variable $energy_{app}$ and is returned at the end of the application execution.

A.0.2 Methodology to Ensure Reliability of Experimental Results

As explained in Section A.0.1, each application is instrumented for measuring its performance and energy consumption. The measured execution times and consumed energy in each run of the application are stored in the variables

te_{cpu}, te_{gpu}, te_{phi}, and $energy_{app}$ respectively, which are returned when the application execution finishes (Sample algorithm 10).

We keep running the application until the sample means of the measured execution times and energy consumption of the application lie within a given confidence interval, and a given precision is achieved. For this, we employ a script, which is named MEANUSINGTTEST. Algorithm 11 presents the pseudocode of this script. It executes the application *app* repeatedly until one of the following three conditions is satisfied:

1. The maximum number of repetitions ($maxReps$) has been exceeded (Line 4).

2. The sample means of all devices (kernel execution times and the application energy consumption) fall in the confidence interval cl, and the precision of measurement eps has been achieved (Lines 11-15).

3. The elapsed time of the repetitions of application execution has exceeded the maximum time allowed ($maxT$ in seconds) (Lines 16-18).

MEANUSINGTTEST returns the sample means of the execution times for each abstract processor (i.e. $time_{cpu}$, $time_{gpu}$, $time_{phi}$) and the energy consumption of all kernels (i.e. $energy$). The input parameters are minimum and maximum number of repetitions, $minReps$ and $maxReps$. These parameter values differ based on the problem size solved. For small problem sizes ($32 \leq n \leq 1024$), these values are set to 10000 and 100000, respectively. For medium problem sizes ($1024 < n \leq 5120$), these values are set to 100 and 1000. For large problem sizes ($n > 5120$), these values are set to 5 and 50. The values of $maxT$, cl, and eps are respectively set to 3600, 0.95, and 0.1. If the precision of measurement is not achieved before the maximum number of repeats have been completed, we increase the number of repetitions and also the maximum elapsed time allowed. However, we observed that condition (2) is always satisfied before the other two in our experiments.

Algorithm 12 shows the pseudocode of the helper functions CALACCURACY, which is used by MEANUSINGTTEST. It returns 1 if the sample mean of

Algorithm 11 Script determining the mean of an experimental run using student's t-test.

1: **procedure** MEANUSINGTTEST($app, minReps, maxReps, maxT, cl, eps, reps\#, elapsedTime, time_{cpu}, time_{gpu}, time_{phi}, energy$)

Input:

The application to execute, app
The minimum number of repetitions, $minReps \in \mathbb{Z}_{>0}$
The maximum number of repetitions, $maxReps \in \mathbb{Z}_{>0}$
The maximum time allowed for the application to run, $maxT \in \mathbb{R}_{>0}$
The required confidence level, $cl \in \mathbb{R}_{>0}$
The required accuracy, $eps \in \mathbb{R}_{>0}$

Output:

The number of experimental runs actually made, $reps\# \in \mathbb{Z}_{>0}$
The elapsed time, $elapsedTime \in \mathbb{R}_{>0}$
The mean execution times, $time_{cpu}, time_{gpu}, time_{phi} \in \mathbb{R}_{\geq 0}$
The mean consumed energy, $energy \in \mathbb{R}_{>0}$

2: $reps \leftarrow 0$; $stop \leftarrow 0$; $etime \leftarrow 0$
3: $sum_{cpu} \leftarrow 0$; $sum_{gpu} \leftarrow 0$; $sum_{phi} \leftarrow 0$; $sum_{eng} \leftarrow 0$
4: **while** ($reps < maxReps$) and (!$stop$) **do**
5: $(t_{cpu}[reps], t_{gpu}[reps], t_{phi}[reps], eng[reps]) \leftarrow$ EXECUTE(app)
6: $sum_{cpu} += t_{cpu}[reps]$
7: $sum_{gpu} += t_{gpu}[reps]$
8: $sum_{phi} += t_{phi}[reps]$
9: $sum_{eng} += eng[reps]$
10: **if** $reps > minReps$ **then**
11: $stop_{cpu} \leftarrow$ CALACCURACY($cl, reps + 1, t_{cpu}, eps$)
12: $stop_{gpu} \leftarrow$ CALACCURACY($cl, reps + 1, t_{gpu}, eps$)
13: $stop_{phi} \leftarrow$ CALACCURACY($cl, reps + 1, t_{phi}, eps$)
14: $stop_{eng} \leftarrow$ CALACCURACY($cl, reps + 1, t_{eng}, eps$)
15: $stop \leftarrow stop_{cpu} \wedge stop_{gpu} \wedge stop_{phi} \wedge stop_{eng}$
16: **if** $\max\{sum_{cpu}, sum_{gpu}, sum_{phi}\} > maxT$ **then**
17: $stop \leftarrow 1$
18: **end if**
19: **end if**
20: $reps \leftarrow reps + 1$
21: **end while**
22: $reps\# \leftarrow reps$;
23: $elapsedTime \leftarrow \max\{sum_{cpu}, sum_{gpu}, sum_{phi}\}$
24: $time_{cpu} \leftarrow \frac{sum_{cpu}}{reps}$; $time_{gpu} \leftarrow \frac{sum_{gpu}}{reps}$; $time_{phi} \leftarrow \frac{sum_{phi}}{reps}$
25: $energy \leftarrow \frac{sum_{eng}}{reps}$
26: **return** $(reps\#, elapsedTime, time_{cpu}, time_{gpu}, time_{phi}, energy)$
27: **end procedure**

a given reading lies in the 95% confidence interval (cl) and a precision of 0.1 ($eps = 10\%$) has been achieved. Otherwise, it returns 0.

Algorithm 12 Algorithm Calculating Accuracy

```
1: function CalAccuracy(cl, reps, Array, eps)
2:     clOut ← fabs(gsl_cdf_tdist_Pinv(cl, reps − 1)) × gsl_stats_sd(Array, 1, reps) / sqrt(reps)
3:     if clOut × reps / (Σ_{i=0}^{reps−1} Array[i]) < eps then
4:         return 1
5:     end if
6:     return 0
7: end function
```

If the precision of measurement is not achieved before the maximum number of repeats have been completed, we increase the number of repetitions and also the maximum elapsed time allowed. However, we observed that condition (2) is always satisfied before the other two in our experiments.

Appendix B

HPOPTA Details

This appendix contains the supporting materials of Chapter 3, “A Novel Data-Partitioning Algorithm for Performance Optimization of Data-Parallel Applications on Heterogeneous HPC Platforms”, which are listed below:

- Comparison of actual and simulated execution times.
- Description of the helper functions used in the algorithm, *HPOPTA*.
- Correctness and complexity proofs of *HPOPTA*.

B.1 Comparison of Actual and Simulated Execution Times

In Section 3.1, we presented the modelling of abstract processors and mentioned that the data points for the same problem size in speed functions are obtained simultaneously. However, during the actual execution of the data-parallel applications using the workload distribution determined by our data partitioning algorithm, the problem sizes executed by the abstract processors can be different. This is because different processors can be allocated different problem sizes by the proposed data partitioning algorithm. Later in the experimental analysis of *HPOPTA*, we evaluated our algorithm and extracted the execution times from the performance profiles.

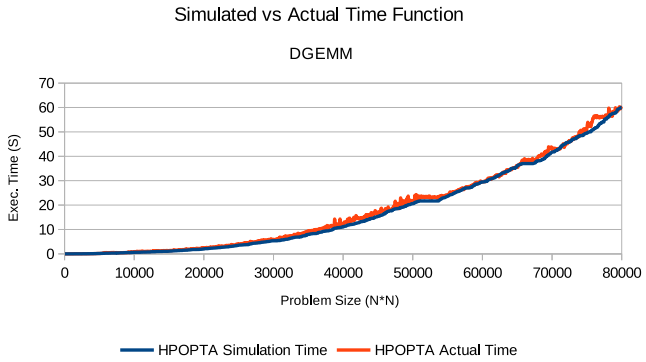

Figure B.1: Comparison of actual with simulated execution times for Matrix Multiplication on HCLServer01*.*

In this section, we experimentally show that the execution times of these problem sizes simultaneously would not differ significantly from those present in the speed functions. We experiment with Matrix Multiplication and 2D FFT, configured for execution on *HCLServer01* as explained in Chapter 3. We compare the execution times of solutions returned by *HPOPTA* with actual execution times on *HCLServer01*. To obtain the actual results, a parallel application is executed where each processor is allocated the problem size given by *HPOPTA*, and its parallel execution time is measured using the experimental methodology explained in Appendix A. Since *HPOPTA* considers all the possible combinations of workload distributions, even combinations where one or more processors are allocated zero workloads, the number of processors used in the experiment ranges from 1 to 3.

We create an experimental data set for Matrix Multiplication including the data points $\{64^2, 128^2, \cdots, 80000^2\}$. Figure B.1 shows the execution times of Matrix Multiplication executed by using *HPOPTA* ($HPOPTA\ Actual\ Time$) compared with the simulated results ($HPOPTA\ Simulation\ Time$).

To analyse FFT, the experimental data set includes data points $\{16^2, 32^2, \cdots, 64000^2\}$. Figure B.2 compares actual with simulated execution times for the application.

From the figures, one can see that the differences between simulated results and actual execution times are insignificant, and they both follow the same trends.

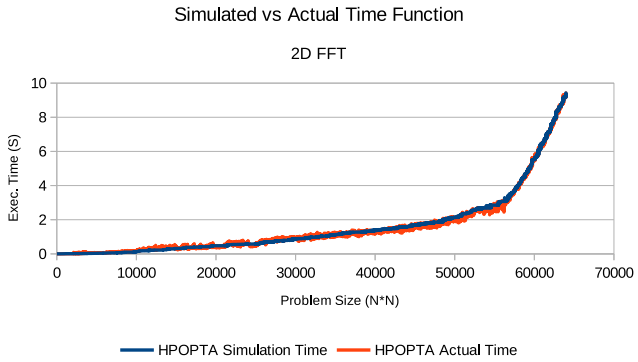

Figure B.2: Comparison of actual with simulated execution times for 2D FFT on HCLServer01*.*

B.2 Helper Routines Called in *HPOPTA*

B.2.1 Function *GetTime*

The function GETTIME(T_i, w) returns the execution time of a given problem size w from the time function T_i (Algorithm 13). It returns 0 when the input problem size w is 0. If there is no match for w in the time function, GETTIME returns -1. It uses time functions which are sorted by problem size, to find the execution times in $O(1)$.

Algorithm 13 Algorithm Finding the Execution Time of a Given Problem Size

```
1: function GETTIME(T_i, w)
2:     if w = 0 then
3:         return 0
4:     end if
5:     if ∄(w, t_iw) ∈ T_i then
6:         return −1
7:     end if
8:     return t_iw
9: end function
```

B.2.2 Function *SizeThresholdCalc*

Algorithm 14 shows the pseudocode of the function SIZETHRESHOLDCALC which calculates size threshold array σ. It, first, determines the size threshold of the last level L_{p-1} (i.e. σ_{p-1}), which is equal to the greatest workload in

the time function T_{p-1} with the execution time less than τ (Line 2). Then, it calculates σ_i, $i \in [0, p-2]$ where σ_i is the summation of σ_{i+1} with the greatest workload in the time function T_i where its execution time is less than τ (Lines 3-5).

Algorithm 14 Algorithm Determining Size Thresholds

1: **function** SIZETHRESHOLDCALC(p, T, τ)
2: $\quad\sigma_{p-1} \leftarrow \max_{j=0}^{m-1}\{x_{(p-1)\ j} \mid t_{(p-1)\ j} < \tau\}$
3: $\quad$**for all** $i = p-2; i \geq 0; i{-}{-}$ **do**
4: $\quad\quad\sigma_i \leftarrow \sigma_{i+1} + \max_{j=0}^{m-1}\{x_{ij} \mid t_{ij} < \tau\}$
5: $\quad$**end for**
6: $\quad$**return** σ
7: **end function**

B.2.3 Function *Cut*

The helper function CUT returns *TRUE* if the workload n is greater than the input size threshold σ (Algorithm 15).

Algorithm 15 Algorithm Cutting Search Tree using the Size Threshold

1: **function** CUT(n, σ)
2: $\quad$**if** $n > \sigma$ **then**
3: $\quad\quad$**return** $TRUE$
4: $\quad$**end if**
5: $\quad$**return** $FALSE$
6: **end function**

B.2.4 Structure of matrix *Mem*

The two-dimensional array *Mem* is used to save solutions for nodes at the levels $\{L_1, \cdots, L_{p-2}\}$ in solution trees. Consider a solution $X = \{x_i, \cdots, x_{p-1}\}$ on processors $\{P_i, \cdots, P_{p-1}\}$ for a given workload $n = \sum_{j=i}^{p-1} x_j$ where $i \in [1, p)$. This solution is memorized in the memory cell $Mem[i][n]$. The memory cell consists of a structure of three fields which are explained below:

- $Mem[i][n].t_{mem}$**:** The parallel execution time of the solution X. It is initialized to the constant value $\_NE$ which means there is no saved solution for the workload n on processors $\{P_i, \cdots, P_{p-1}\}$.

- $Mem[i][n].idx_{last}$**:** The index of the last data point in the time function T_i which has been examined yet. This field helps *HPOPTA* to resume the examination of further points from where it was interrupted by the operation *Backtrack*. In addition, idx_{last} is used to label a memory cell as *Finalized* by setting idx_{last} to the constant value $\_FI$. A finalized memory cell contains an optimal solution.

- $Mem[i][n].x_{mem}$**:** The problem size which is assigned to P_i by the solution X (i.e. x_i).

B.2.5 Function *ReadMemory*

Algorithm 16 illustrates the function READMEMORY. This function retrieves a saved solution for a given workload n, executing on processors $\{P_c, \cdots, P_{p-1}\}$ where $c \in [1, p-1)$. The retrieved solution is stored in X_{cur}. READMEMORY, first, reads $Mem[c][n]$ to retrieve the parallel execution time of the workload n on the processors $\{P_c, \cdots, P_{p-1}\}$ and the problem size given to P_c (Algorithm 16, Line 2). According to the retrieved values of $Mem[c][n].t_{mem}$ and $Mem[c][n].idx_{last}$, the following cases might happen:

- **NOT_SOLUTION**: This case happens when there is no saved solution in $Mem[c][n]$ (t_{mem} is $\_NE$), and the memory cell has been finalized ($Mem[c][n].idx_{last}$ is $\_FI$). It means that there is no solution for n on processors $\{P_c, \cdots, P_{p-1}\}$ (Lines 4-5).

- **SOLUTION**: This case occurs when there is a finalized solution for n on processors $\{P_c, \cdots, P_{p-1}\}$. In this case, $X_{cur} = \{x_{cur}[0], \cdots, x_{cur}[c-1], \cdots, x_{cur}[p-1]\}$ determines the retrieved solution from memory. However, if the execution time of the saved solution is greater than τ ($t_{mem} > \tau$) the saved solution is ignored, and $NOT\_SOLUTION$ is returned (Lines 6-14).

- **SOLUTION_RESUME**: This case happens when there is a memorized solution in $Mem[c][n]$, but it is not finalized. The function READMEMORY

retrieves the saved solution for n on processors $\{P_c, \cdots, P_{p-1}\}$ and invokes PROCESSSOLUTION to process it. READMEMORY also sets idx to the retrieved $Mem[c][n].idx_{last}$ to make *HPOPTA_Kernel* resume the examination of further points from the idx_{last}-th data point in the time function T_c, rather than starting from the beginning (Lines 15-22).

- **RESUME**: This case occurs when there is no solution for the workload n on processors $\{P_c, \cdots, P_{p-1}\}$, and *HPOPTA_Kernel* will resume the tree exploration form $Mem[c][n].idx_{last}$. (Lines 23-26).

Algorithm 16 Algorithm Retrieving Solution from Memory

```
1: function READMEMORY(n, p, c, τ, T, X_cur, Mem, idx)
2:     ⟨t_mem, idx_last, x_mem⟩ ← Mem[c][n]
3:     if idx_last = _FI then
4:         if t_mem = _NE then
5:             return NOT_SOLUTION
6:         else
7:             if t_mem < τ then
8:                 x_cur[c] ← x_mem
9:                 x_cur[i] ← Mem[i][n − Σ_{j=c}^{i−1} x_cur[j]].x_mem, ∀i ∈ [c + 1, p − 2]
10:                x_cur[p − 1] ← n − Σ_{i=c}^{p−2} x_cur[i]
11:                return SOLUTION
12:            end if
13:            return NOT_SOLUTION
14:        end if
15:    else if idx_last ≠ _FI then
16:        if t_mem ≠ _NE ∧ t_mem < τ ∧ x_{c idx_last} ≠ x_mem then
17:            x_cur[c] ← x_mem
18:            x_cur[i] ← Mem[i][n − Σ_{j=c}^{i−1} x_cur[j]].x_mem, ∀i ∈ [c + 1, p − 2]
19:            x_cur[p − 1] ← n − Σ_{i=c}^{p−2} x_cur[i]
20:            idx ← idx_last
21:            return SOLUTION_RESUME
22:        end if
23:        if idx_last ≠ _NE then
24:            idx ← idx_last
25:            return RESUME
26:        end if
27:    end if
28:    return DUMMY
29: end function
```

B.2.6 Function *ProcessSolution*

The function PROCESSSOLUTION (Algorithm 17) is invoked once a solution is found. The found solution is determined by $X_{cur} = \{x_{cur}[0], \cdots, x_{cur}[p-1]\}$. The input parameter $m_{idx} \in \{1, 2, \cdots, p-2\}$ determines some part of the

solution, including problem sizes $\{x_{cur}[m_{idx}], \cdots, x_{cur}[p-1]\}$, which has been retrieved from Mem. To process a solution, the function first stores it in Mem (Lines 5-14). It then updates three variables σ, τ and X_{opt} if the execution time of the new solution is less than current τ (Lines 20-23). In the case of equal execution times for solutions in X_{cur} and X_{opt}, X_{opt} is updated to the solution with less active processors (processors with non-zero workloads) (Lines 24-38).

Algorithm 17 Algorithm Processing the Found Solution

```
1: function ProcessSolution(p, T, τ, σ, bk, X_cur, Mem, m_idx, X_opt)
2:     sum_size ← x_cur[p − 1]
3:     t_max ← GetTime(T_{p−1}, x_cur[p − 1])
4:     idx_max ← p − 1
5:     for i = p − 2, i ≥ 1, i − − do
6:         if GetTime(T_i, x_cur[i]) ≥ t_max then
7:             t_max ← GetTime(T_i, x_cur[i])
8:             idx_max ← i
9:         end if
10:        sum_size ← sum_size + x_cur[i]
11:        if sum_size ≠ 0 ∧ i < m_idx then
12:            Save(i, sum_size, t_max, x_cur[i], Mem)
13:        end if
14:    end for
15:    if GetTime(T_0, x_cur[0]) ≥ t_max then
16:        t_max ← GetTime(T_0, x_cur[0])
17:        idx_max ← 0
18:    end if
19:    bk ← idx_max
20:    if τ > t_max then
21:        τ ← t_max
22:        X_opt ← X_cur
23:        σ ← SizeThresholdCalc(p, T, τ)
24:    else if τ = t_max then
25:        proc_cur ← 0
26:        proc_last ← 0
27:        for i = 0, i < p, i + + do
28:            if x_cur[i] ≠ 0 then
29:                proc_cur + +
30:            end if
31:            if x_opt[i] ≠ 0 then
32:                proc_last + +
33:            end if
34:        end for
35:        if proc_cur < proc_last then
36:            X_opt ← X_cur
37:        end if
38:    end if
39: end function
```

B.2.7 Function *Save*

Algorithm 18 illustrates the function SAVE which memorizes all solutions found during the tree exploration. Consider a found solution for a given workload n on processors $\{P_i, \cdots, P_{p-1}\}$ where $i \in \{1, 2, \cdots, p-2\}$. The input parameter t_{max} determines the parallel execution time of the solution, and x is the problem size allocated to P_i. The function will update no memory cell if the newly found distribution has a greater parallel execution time than that of the saved one.

Algorithm 18 Algorithm Storing Solutions into Matrix Mem

```
1: function SAVE(i, n, t_max, x, Mem)
2:     if Mem[i][n].t_mem = _NE ∨ Mem[i][n].t_mem > t_max then
3:         Mem[i][n] ← ⟨t_max, −, x⟩
4:     end if
5: end function
```

B.2.8 Function *Backtrack*

Algorithm 19 shows the pseudocode of BACKTRACK which is called by *HPOPTA_Kernel* to make a decision if the recursive tree exploration process backtracks from current node to its upper node at the level bk (i.e. L_{bk}) or not. The function returns *TRUE* in the case of backtracking. In addition, this function performs memory finalization and stores resuming points by setting idx_{last} in the corresponding memory cells to $\_FI$.

B.2.9 Function *MakeFinal*

Algorithm 20 illustrates the function MAKEFINAL which is responsible to finalize a given memory cell mem. It sets $mem.idx_{last}$ to the constant value $\_FI$. The finalized memory cell contains the optimal distribution.

Algorithm 19 Algorithm Implementing Backtracking and Matrix Mem Cell Finalization

```
1: function BACKTRACK(n, c, bk, idx, t_cur, τ, Mem, isMem)
2:     if bk < c then
3:         if isMem = TRUE then
4:             return TRUE
5:         end if
6:         if t_cur = τ then
7:             Mem[c][n].idx_last ← _FI
8:         else
9:             Mem[c][n].idx_last ← idx
10:        end if
11:        return TRUE
12:    else if bk = c then
13:        bk ← NULL
14:        Mem[c][n].idx_last ← _FI
15:        return TRUE
16:    else
17:        bk ← NULL
18:        return FALSE
19:    end if
20: end function
```

Algorithm 20 Algorithm Finalizing a Matrix Mem Cell

```
1: function MAKEFINAL(mem)
2:     mem.idx_last ← _FI
3: end function
```

B.3 Correctness Proof of *HPOPTA*

Proposition B.3.1. *The algorithm* HPOPTA *always returns an optimal distribution for a given workload n between p heterogeneous processors which minimizes its parallel execution time.*

Proof. To find an optimal distribution for a workload n executing on processors $\{P_0, \cdots, P_{p-1}\}$ with minimum execution time, the straightforward approach is to expand the full search tree and examine all possible distributions. This approach has, however, an exponential complexity. Rather than building all solutions, *HPOPTA* only builds and explores a small fraction of the full search tree. To achieve this, it applies two specific operations, *Cut* and *Backtrack*, that remove subtrees of the full search tree without their construction and exploration.

Therefore, the correctness of *HPOPTA* will be proved if we show that no subtree removed by *HPOPTA* from consideration contains a solution, superior to the one finally returned by *HPOPTA*. We will prove this below:

Cut: Consider a given node at level L_i which is labelled by n. Operation *Cut* will remove the subtree, growing from the node, only if the workload n is greater than the corresponding size threshold of this node, which is σ_i. Remember that σ_i represents the maximum workload size that can be executed in parallel by processors $\{P_i, \cdots, P_{p-1}\}$ faster than τ time units, where τ is the execution time of the fastest solution which is found so far. Therefore, the parallel execution time of any solution in the removed subtree cannot be less than τ and hence less than the time of the globally fastest solution.

Backtrack: This operation is only applied when *HPOPTA* finds a solution like $\{x_0, x_1, \cdots, x_k, \cdots, x_{p-1}\}$ such that $\max_{i=0}^{p-1} t_i(x_i) = t_k(x_k) = \tau$. We know two facts:

1. The parallel execution time of any solution involving a subtree with the root node x_k cannot be less than τ. Therefore, this subtree can be ignored.

2. *HPOPTA* arranges nodes at every level of the search tree in non-decreasing order of execution time. Therefore, all nodes at the level L_k which are placed after the node x_k will have execution times greater than or equal to $t_k(x_k)$. Therefore, expansion of these nodes cannot lead to a solution which has an execution time less than τ.

From these facts, we can conclude that the construction and examination of the subtrees, which would grow from the node x_k and the following nodes at level L_k of the search tree, will not result in a distribution with the execution time less than τ and can be ignored by backtracking to the upper level L_{k-1}. *End of Proof.*

B.4 Complexity of *HPOPTA*

Lemma B.4.1. *The complexity of* HPOPTA_Kernel *is* $O(m^3 \times p^3)$.

Proof. HPOPTA_Kernel is an optimal recursive algorithm for exploring solution trees. Thus, its time complexity can be expressed in terms of the number

of its recursive invocations. We formulate the number of recursive calls in each level of the solution tree using a trivial example.

Consider a workload n executing on a platform consisting of 5 heterogeneous processors ($p = 5$). Suppose the time function of each processor contains 2 data points ($m = 2$) where $t_i(x) = \{(\Delta x, t_i(\Delta x)), (2\Delta x, t_i(2\Delta x))\}$, where $\Delta x \in \mathbb{N}$ is the minimum granularity of problem sizes in time functions and $i \in \{0, 1\}$. It should be mentioned that there is no assumption on the value of Δx, and it, therefore, can be used without loss of generality. For the sake of simplicity, we assume that execution time increases with increasing in problem size in time functions. That is, the data points in time functions are visited in the increasing order of problem size. This assumption does not apparently make the proof less general. However, it makes finding the formula for complexity less difficult.

Figure B.3 shows the solution tree of this example. Since we are looking for an upper bound for the time complexity of *HPOPTA_Kernel*, we consider n greater than $8\Delta x$, which is the maximum problem size subtracted from n in the figure. For the sake of simplicity, only the operation *Save* is considered. Other optimizations, *Cut* and *Backtrack*, are not applied. That is why the practical time complexity of *HPOPTA* is enormously less than the theoretical one. In the figure, nodes highlighted in red have already been expanded in the same level and their solutions are retrieved from matrix Mem instead of node expansions.

The number of recursive calls in each level of tree (nodes in black) can be calculated as follows:

$$C\#(L) = \begin{cases} L \times m + 1 & 0 \leq L < p - 1 \\ C\#(p-2) \times (m+1) & L = p - 1 \end{cases}$$

where L represents the level number in the tree.

Therefore, $C\#(L)$ is equal to:

$$C\#(L) = \begin{cases} L \times m + 1 & 0 \leq L < p - 1 \\ m^2 \times p - 2 \times m^2 + m \times p - m + 1 & L = p - 1 \end{cases}$$

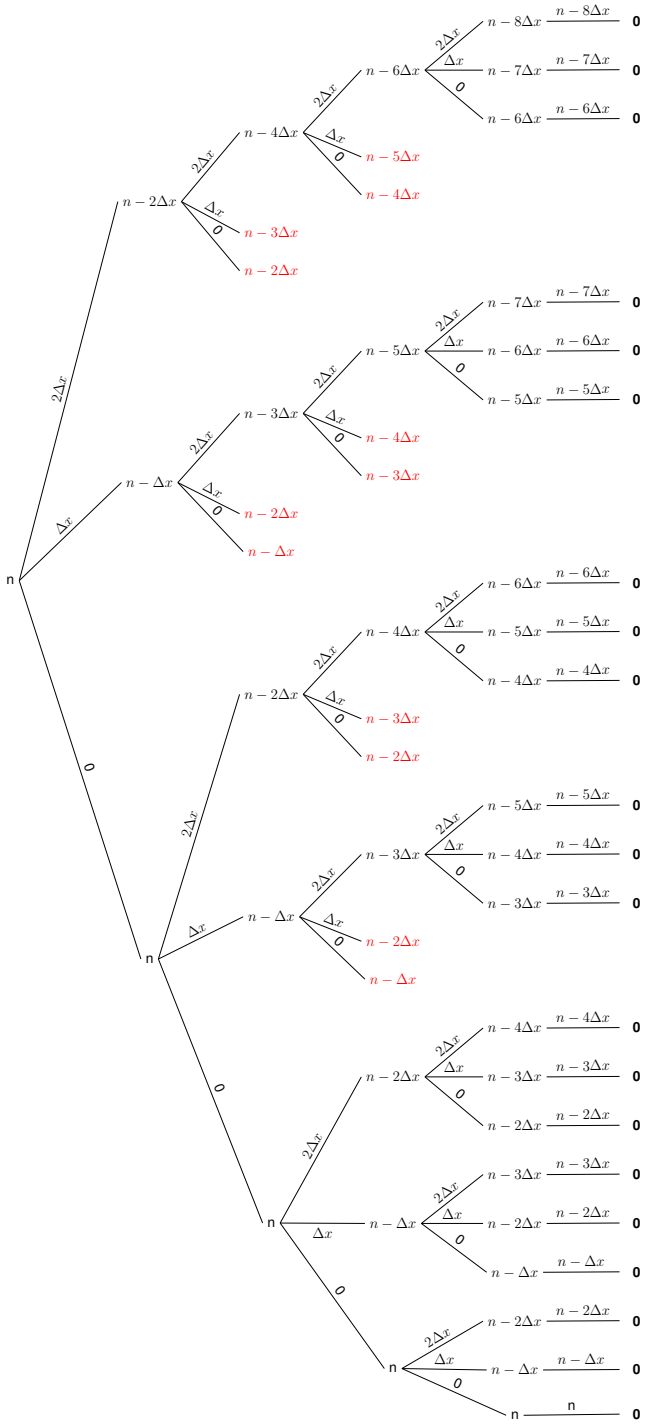

Figure B.3: The execution of HPOPTA *for a sample set of time functions ($p = 5$), each contains 2 data points. The memorization technique is only considered to reduce the full search space of solutions. The other optimizations, time threshold, size threshold, and backtracking, are not applied.*

That is, the total number of recursive calls for *HPOPTA_Kernel* is equal to $\sum_{L=0}^{p-1}(C\#(L))$ which has order of $O(m \times p^2 + m^2 \times p)$.

In addition, the number of nodes in each level that their solutions are retrieved from matrix Mem is equal to:

$$\begin{aligned} \text{Memory\#(L)} &= (C\#(L-1)-1) \times m \\ &= (m^2) \times (L-1), \\ & 1 \leq L \leq p-2 \end{aligned}$$

To retrieve saved results from Mem, the function READMEMORY reads up to $p-2$ elements from Mem. That is, the complexity of READMEMORY is $O(p)$. We know *HPOPTA_Kernel* accesses matrix Mem for levels 1 to $p-2$. Therefore, the cost of all READMEMORY invocations to find solutions is equal to $\sum_{L=1}^{p-2}(Memory\#(L) \times O(p))$, which is equal to $O(m^2 \times p^3)$.

Furthermore, there are solutions which are found in the last level of the tree (For instance level L_4 in the Figure B.3). For these solutions, *HPOPTA_Kernel* reads time function $T_{p-1}(x)$ with the time complexity of $O(1)$ to find the execution time of the given workload to P_{p-1}. The number of nodes in the level L_{p-1} is equal to $C\#(p-1)$ nodes. That is, the cost of finding solutions in the last level of the tree is $O(m^2 \times p)$.

Once a solution is found, PROCESSSOLUTION is invoked with a computational complexity of $O(m \times p)$. In the worst case, it is invoked for each leaf of the tree, either after each call of READMEMORY or after finding a solution in level L_{p-1}. Therefore, an upper bound for all PROCESSSOLUTION calls is $(C\#(p-1) + \sum_{L=1}^{p-2} Memory\#(L)) \times O(m \times p) = O(m^3 \times p^3)$.

The worst time complexity of *HPOPTA_Kernel* can therefore be summarized as follows:

$$\begin{aligned} \text{Complexity(}\textit{HPOPTA\_Kernel}\text{)} = & \text{O(recursive calls of } \textit{HPOPTA\_Kernel}\text{)} + \\ & \text{O(READMEMORY calls)} + \\ & \text{O(finding solutions in } L_{p-1}\text{)} + \\ & \text{O(PROCESSSOLUTION calls).} \end{aligned}$$

which equals:

$$\begin{aligned}\text{Complexity(}\textit{HPOPTA\_Kernel}\text{)} =& O(m \times p^2 + m^2 \times p) + \\ & O(m^2 \times p^3) + \\ & O(m^2 \times p) + \\ & O(m^3 \times p^3) \\ & = O(m^3 \times p^3).\end{aligned}$$

Proposition B.4.2. *The complexity of* HPOPTA *is* $O(m^3 \times p^3)$.

Proof. HPOPTA consists of following main steps:

- **Sorting:** There are p discrete time functions with a cardinality of m. These functions can be sorted in the non-decreasing order of execution time with a computational complexity of $O(p \times m \times \log_2 m)$.
- **Finding load–equal distribution and initialization of time threshold:** This step has complexity $O(p)$.
- **Finding size thresholds:** To find the size threshold of a given level L_i, $i \in [0, p-1]$, all data points, existing in $t_i(x)$, with execution times less than or equal to τ should be examined. This process has a complexity of $O(m)$. Therefore, finding p size thresholds has a complexity of $O(p \times m)$.
- **Memory initialization:** In this step, all $(n+1) \times (p-2)$ cells of Mem are initialized with a complexity of $O(n \times p)$.
- **Kernel invocation:** According to the lemma B.4.1, the complexity of this step is $O(m^3 \times p^3)$.

Therefore, one can conclude a computational complexity of *HPOPTA* is equal to the summation of all these steps, which is $O(m^3 \times p^3)$. *End of Proof.*

Proposition B.4.3. *The total memory used by the algorithm is* $O(p \times (m+n))$.

Proof. HPOPTA uses memory to store following items:

- **time functions:** There exist p discrete time functions, with a cardinality of m. We store both size-sorted (sorted by problem size) and time-sorted (sorted by execution time) functions. This requires $2 \times p \times m$ memory cells.

- X_{cur}**:** This array stores the problem sizes assigned to each processor by the current solution. That is, this is an array of size p cells.

- X_{opt}**:** This array stores the problems sizes assigned to each processor by the optimal solution found so far. That is, this is an array of size p cells.

- **Mem:** This is a matrix consisting of $(p-2) \times (n+1)$ cells.

Thus, total memory cells used by *HPOPTA* is equal to $2 \times m \times p + 2 \times p + (p-2) \times (n+1) \approxeq O(p \times (m+n))$. *End of Proof.*

Appendix C

HEOPTA Details

In this chapter, we, first, explain the supporting materials of Chapter 4, "A Novel Model-based Algorithm for Dynamic Energy Consumption Optimization of Data-Parallel Applications on Heterogeneous HPC Platforms". We then prove the correctness and complexity of the proposed algorithm, *HEOPTA*.

C.1 Helper Routines Called in *HEOPTA*

C.1.1 Function *GetEng*

The function GETENG(E_i, w) returns the dynamic energy consumption of a given problem size w from the energy function E_i (Algorithm 21). If there is no data point in E_i with the problems size of w, the function returns -1. It returns 0 for zero problem sizes. This function uses dynamic energy functions which are sorted by problem size. This facilitates finding output results in $O(1)$.

Algorithm 21 Algorithm Extracting the Consumed Energy of a Given Problem Size

```
1: function GETENG(E_i, w)
2:     if w = 0 then
3:         return 0
4:     end if
5:     if ∄(w, e_iw) ∈ E_i then
6:         return −1
7:     end if
8:     return e_iw
9: end function
```

C.1.2 Function *SizeThresholdCalc*

Algorithm 22 presents the pseudocode of SIZETHRESHOLDCALC which determines size threshold array σ. The function calculates the size threshold of L_{p-1} which is equal to the greatest workload in the energy function E_{p-1} that its dynamic energy consumption is less than ε (Line 2). It then calculates σ_i, $i \in \{0, 1, , \cdots, p-2\}$ which is the summation of σ_{i+1} with the greatest workload in the function E_i where its dynamic energy consumption is less than ε (Lines 3-5).

Algorithm 22 Algorithm Determining Size Thresholds

1: **function** SIZETHRESHOLDCALC(p, E, ε)
2: $\quad\sigma_{p-1} \leftarrow \max_{j=0}^{m-1}\{x_{(p-1)\ j} \mid e_{(p-1)\ j} < \varepsilon\}$
3: $\quad$**for all** $i = p-2; i \geq 0; i{-}{-}$ **do**
4: $\quad\quad\sigma_i \leftarrow \sigma_{i+1} + \max_{j=0}^{m-1}\{x_{ij} \mid e_{ij} < \varepsilon\}$
5: $\quad$**end for**
6: $\quad$**return** σ
7: **end function**

C.1.3 Function *Cut*

The pseudocode of the function CUT is presented in Algorithm 23. It returns *TRUE* if the workload n is greater than the input size threshold σ.

Algorithm 23 Algorithm Cutting Search Tree using the Size Threshold

1: **function** CUT(n, σ)
2: $\quad$**if** $n > \sigma$ **then**
3: $\quad\quad$**return** $TRUE$
4: $\quad$**end if**
5: $\quad$**return** $FALSE$
6: **end function**

C.1.4 Structure of matrix *Mem*

The two dimensional matrix Mem memorizes solutions for the nodes visited at levels $\{L_1, \cdots, L_{p-2}\}$ in solution trees. Consider a given distribution $X = \{x_i, \cdots, x_{p-1}\}$ as a solution for a workload n on processors $\{P_i, \cdots, P_{p-1}\}$ where $i \in \{1, 2, \cdots, p-1\}$. The solution is saved in $Mem[i][n]$, which is a memory cell consisting of two fields:

- $Mem[i][n].e_{mem}$**:** The amount of dynamic energy consumption for the parallel execution of X running on processor $\{P_i, \cdots, P_{p-1}\}$. It is initialized to the constant value $\_NE$ implying the node n in the solution tree has not been explored, and therefore, there is no saved solution for it. If expansion of node n at level L_i ends in no solution, its corresponding memory cell, $Mem[i][n].e_{mem}$, is set to $\_NS$.

- $Mem[i][n].x_{mem}$**:** The problem size is allocated to P_i by the distribution X.

C.1.5 Function *ReadMemory*

Function READMEMORY retrieves the saved solution for a given workload n on processors $\{P_c, \cdots, P_{p-1}\}$ where $c \in \{1, 2, \cdots, p-2\}$ (Algorithm 24). According to the retrieved value for $Mem[c][n].e_{mem}$, the following cases might happen:

- **NOT_SOLUTION**: This case occurs when either e_{mem} is equal to $\_NS$ (there is no solution for n on the processor $\{P_c, \cdots, P_{p-1}\}$) or the dynamic energy consumption of the saved distribution (e_{mem}) is greater than or equal to ε (Lines 7 and 11).

- **SOLUTION**: This case takes place if e_{mem} is not equal to $\_NE$ or $\_NS$, and its value is also less than ε (Lines 13-16).

Finally, the output array X_{cur} determines the retrieved solution.

C.1.6 Function *ProcessSolution*

Algorithm 25 presents the pseudocode of PROCESSSOLUTION. It is invoked after finding any solution. First, it saves the solution, which is determined by $X_{cur} = \{x_{cur}[0], \cdots, x_{cur}[p-1]\}$, in Mem (Lines 4-13). The input parameter $m_{idx} \in \{1, 2, \cdots, p-2\}$ determines some part of the solution, including problem sizes $\{x_{cur}[m_{idx}], \cdots, x_{cur}[p-1]\}$, which has been read from Mem. If the amount of dynamic energy which is consumed by the solution X is less

Algorithm 24 Algorithm Retrieving Solution from Memory

```
1: function READMEMORY(n, p, c, ε, E, X_cur, Mem)
2:     ⟨e_mem, x_mem⟩ ← Mem[c][n]
3:     if e_mem = _NE then
4:         return DUMMY
5:     end if
6:     if e_mem = _NS then
7:         return NOT_SOLUTION
8:     end if
9:     if e_mem ≠ _NE then
10:        if e_mem ≥ ε then
11:            return NOT_SOLUTION
12:        end if
13:        x_cur[c] ← x_mem
14:        x_cur[i] ← Mem[i][n − Σ_{j=c}^{i−1} x_cur[j]].x_mem, ∀i ∈ [c + 1, p − 2]
15:        x_cur[p − 1] ← n − Σ_{i=c}^{p−2} x_cur[i]
16:        return SOLUTION
17:    end if
18: end function
```

than current ε then X_{opt}, array σ and ε are updated (Lines 15-18). If dynamic energy consumption for the current solution is equal to ε then X_{opt} will be updated to the current solution if the number of idle processors in current solution is greater than that of X_{opt} (Lines 19-33). It should be mentioned that idle processors are those with zero workloads.

C.1.7 Function *Save*

Algorithm 26 illustrates the function SAVE which memorizes all found solutions. Consider a solution for a given workload n on processors $\{P_i, \cdots, P_{p-1}\}$, $i \in \{1, 2, \cdots, p-2\}$, with a dynamic energy consumption of sum_{eng}. The solution allocates a problem size x to P_i. The function will update no memory cell if the newly found distribution has greater consumed energy than that of the saved solution.

C.1.8 Function *MakeFinal*

Algorithm 27 shows the function MAKEFINAL which finalizes a given memory cell mem. It sets $mem.e_{mem}$ to the constant value $\_NS$ if it is equal to $\_NE$. The finalized memory cell contains the optimal distribution. Setting the e_{mem} of a memory cell to $\_NS$ means that its corresponding node in the solution

Algorithm 25 Algorithm Processing the Found Solution

```
1: function ProcessSolution(p, E, ε, σ, X_cur, Mem, m_idx, X_opt)
2:     sum_size ← x_cur[p − 1]
3:     sum_eng ← GetEng(E_{p−1}, x_cur[p − 1])
4:     for i = p − 2, i ≥ 1, i − − do
5:         sum_size ← sum_size + x_cur[i]
6:         sum_eng ← sum_eng + GetEng(E_i, x_cur[i])
7:         if sum_eng > ε then
8:             return
9:         end if
10:        if sum_size ≠ 0 ∧ i < m_idx then
11:            Save(i, sum_size, sum_eng, x_cur[i], Mem)
12:        end if
13:    end for
14:    sum_eng ← sum_eng +GetEng(E_0, x_cur[0])
15:    if ε > sum_eng then
16:        ε ← sum_eng
17:        X_opt ← X_cur
18:        σ ← SizeThresholdCalc(p, E, ε)
19:    else if ε = sum_eng then
20:        proc_cur ← 0
21:        proc_last ← 0
22:        for i = 0, i < p, i + + do
23:            if x_cur[i] ≠ 0 then
24:                proc_cur + +
25:            end if
26:            if x_opt[i] ≠ 0 then
27:                proc_last + +
28:            end if
29:        end for
30:        if proc_cur < proc_last then
31:            X_opt ← X_cur
32:        end if
33:    end if
34: end function
```

Algorithm 26 Algorithm Storing Solutions into Matrix Mem

```
1: function Save(i, n, sum_eng, x, Mem)
2:     if Mem[i][n].e_mem = _NE ∨ Mem[i][n].e_mem > sum_eng then
3:         Mem[i][n] ← ⟨sum_eng, x⟩
4:     end if
5: end function
```

tree has been explored with no solution.

Algorithm 27 Algorithm Finalizing a Matrix Mem Cell

```
1: function MAKEFINAL(mem)
2:     if mem.e_mem = _NE then
3:         mem.e_mem ← _NS
4:     end if
5: end function
```

C.2 Correctness Proof of *HEOPTA*

Proposition C.2.1. *The algorithm* HEOPTA *always returns an optimal solution for a given workload n running on p heterogeneous processors which minimizes its parallel dynamic energy consumption.*

Proof. The algorithm *HEOPTA* is based on the naive approach, which explores the full solution tree to build all possible distributions. However, *HEOPTA* employs the specific operation *Cut* to just build a smaller fraction of the full solution tree and reduce the computational complexity from exponential to polynomial time. Therefore, the correctness of *HEOPTA* will be proved if we show that there exists no subtree which is cut by the operation *Cut* while contains a solution superior to the one eventually returned by *HEOPTA*.

Consider a given node at level L_i which is labelled by n. The operation *Cut* will remove the subtree, growing from the node, only if the workload n is greater than the size threshold of this node, which is σ_i. Remember that σ_i represents the maximum workload can be executed in parallel on processors $\{P_i, \cdots, P_{p-1}\}$ so that the dynamic energy consumption by every processor $\{P_i, \cdots, P_{p-1}\}$ is less than ε. Therefore, the dynamic energy consumption of any solution in the removed subtree cannot be less than ε, and hence, less than that of the globally optimal solution which is returned by *HEOPTA*. *End of Proof*.

C.3 Complexity of *HEOPTA*

Lemma C.3.1. *The time complexity of* HEOPTA_Kernel *is* $O(m^3 \times p^3)$.

Proof. HEOPTA_KERNEL is an efficient recursive algorithm for solution tree exploration. Thus, its computational complexity can be related in terms of the number of recursive invocations. We use a trivial solution tree to formulate the number of recursive calls in each level.

Given is a workload n running on 5 heterogeneous processors ($p = 5$). Suppose there are 5 discrete dynamic energy functions $E = \{e_0(x), \cdots, e_{p-1}(x)\}$ with the cardinality of 2 ($m = 2$) where $e_i(x) = \{(\Delta x, e_i(\Delta x)), (2\Delta x, e_i(2\Delta x))\}$, $i \in \{0, 1\}$, and Δx represents the minimum granularity of workload in the functions. It should be noted that HEOPTA_KERNEL is capable of dealing with any granularity, therefore, this assumption does not make the proof less general. For the sake of simplicity, suppose the energy consumption increase as workload size increases in energy functions. This assumption does not make the proof less general. However, it eases finding the formula.

Figure C.1 illustrates the solution tree for finding the optimal distribution for the workload size n on the 5 processors. We consider n greater than $8\Delta x$, the maximum problem size subtracted from n in this example. This assumption ensures finding the upper bound for HEOPTA_KERNEL. Because of only considering the operation SAVE to obtain the complexity of *HEOPTA*, its practical time complexity is considerably less than the theoretical one. In the Figure, red nodes represent ones have been already expanded in the same level, and their solutions are retrieved from the matrix Mem rather than node expansion.

Eq. C.1 formulates the number of recursion calls in each level of the solution tree explored by HEOPTA_KERNEL.

$$C\#(L) = \begin{cases} L \times m + 1 & 0 \leq L < p - 1 \\ C\#(p-2) \times (m+1) & L = p - 1 \end{cases} \tag{C.1}$$

where L represents the level number.

Eq. C.2 shows the expanded form of Eq. C.1.

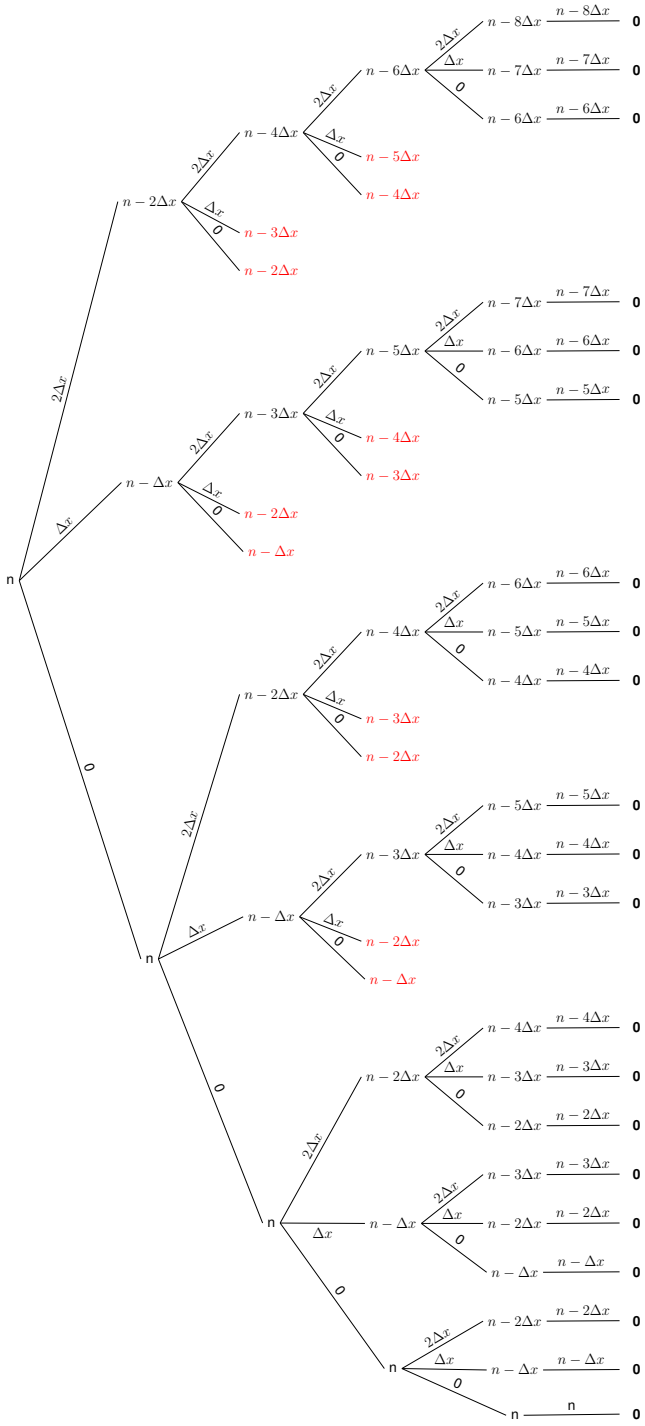

Figure C.1: The solution tree representing the execution of HEOPTA *for a sample set of energy functions ($p = 5$), each contains two data points. The memorization technique is only considered to reduce the full search space of solutions. The other optimization,* cut*, is not applied.*

$$C\#(L) = \begin{cases} L \times m + 1 & 0 \leq L < p-1 \\ m^2 \times p - 2 \times m^2 + m \times p - m + 1 & L = p-1 \end{cases} \quad \text{(C.2)}$$

That is, the total number of recursive calls is equal to $\sum_{L=0}^{p-1}(C\#(L))$ which has complexity of $O(m \times p^2 + m^2 \times p)$.

In addition, the number of nodes in each level that their solutions are retrieved from Mem is formulated in Eq. C.3.

$$\begin{aligned} \text{Memory\#(L)} &= (C\#(L-1) - 1) \times m \\ &= (m^2) \times (L-1), \\ & 1 \leq L \leq p-2 \end{aligned} \quad \text{(C.3)}$$

Eq. C.3 determines the number of nodes in each level that their solutions retried form Mem (nodes highlighted in red in Figure C.1). Since solutions found on levels 1 to $p-2$ are memorized, total number of red nodes would be $\sum_{L=1}^{p-2} Memory\#(L) = O(m^2 \times p^2)$. To retrieve a solution from Mem, the function READMEMORY is invoked which reads up to $p-2$ elements from Mem. That is, READMEMORY has a complexity of $O(p)$. Therefore, the computational cost for finding solutions using Mem is equal to $O(m^2 \times p^3)$.

Apart from solutions which are found using Mem, there are other solutions which are found in the last level of the tree (For instance level L_4 in the Figure C.1). For this type of solutions, *HEOPTA_Kernel* reads time function $T_{p-1}(x)$ with the time complexity of $O(1)$ to obtain the dynamic energy consumption of the given workload to P_{p-1}. The number of nodes in the level L_{p-1} is equal to $C\#(p-1)$ nodes. That is, the cost of finding solutions in the last level of the tree is $O(m^2 \times p)$.

Once a solution is found, PROCESSSOLUTION is invoked with a complexity of $O(m \times p)$. In the worst case, it is invoked when either a solution is found using Mem (after each call of READMEMORY) or a solution is found in the a leave. Therefore, the total complexity of all PROCESSSOLUTION calls is equal to $(C\#(p-1) + \sum_{L=1}^{p-2} Memory\#(L)) \times O(m \times p) = O(m^3 \times p^3)$.

The time complexity of HEOPTA_KERNEL can be calculated as follows:

$$\begin{aligned}\text{Complexity(HEOPTA\_KERNEL)} =& \text{O(recursive calls of HEOPTA\_KERNEL)}+\\ & \text{O(ReadMemory cost)}+\\ & \text{O(findng solutions in } L_{p-1})+\\ & \text{O(PROCESSSOLUTION calls).}\end{aligned}$$

which equals:

$$\begin{aligned}\text{Complexity(}\textit{HEOPTA\_Kernel}\text{)} =& O(m \times p^2 + m^2 \times p)+\\ & O(m^2 \times p^3)+\\ & O(m^2 \times p)+\\ & O(m^3 \times p^3)\\ & = O(m^3 \times p^3).\end{aligned}$$

Proposition C.3.2. *The computational complexity of* HEOPTA *is* $O(m^3 \times p^3)$.

Proof. HEOPTA consists of following main steps:

- **Sorting:** There exist p discrete energy functions with cardinality of m. The complexity to sort all of them in the non-decreasing order of consumed energy is $O(p \times m \times \log_2 m)$.
- **Finding load-equal distribution and initialization of energy threshold:** This step has complexity $O(p)$.
- **Finding size thresholds:** To find size threshold for a given level L_i, $i \in [0, p-1]$, all data points, existing in $e_i(x)$, with consumed energies less than ε should be examined. This has complexity of $O(m)$. Therefore, finding p size thresholds has a complexity of $O(p \times m)$.
- **Memory initialization:** In this step, all $(n+1) \times (p-2)$ cells of Mem are initialized with a complexity of $O(n \times p)$.
- **Kernel invocation:** According to the lemma C.3.1, the complexity of this step is $O(m^3 \times p^3)$.

Therefore, one can conclude that the time complexity of *HEOPTA* equals the summation of all these complexities, which is equal to $O(m^3 \times p^3)$. *End of Proof*.

Proposition C.3.3. *The total memory used by the algorithm is* $O(p \times (m+n))$.

Proof. HEOPTA uses memory to store following information:

- **energy functions:** There are p discrete dynamic energy functions with the cardinality of m. We stores both size-sorted (sorted by problem size) and energy-sorted (sorted by dynamic energy consumption). Therefore, $2 \times p \times m$ memory cells are required for storing these functions.
- X_{cur}**:** This array stores the problem sizes assigned to each processor by the current solution. That is, this is an array of size p cells.
- X_{opt}**:** This array stores the problem sizes assigned to each processor by the optimal solution found so far. That is, this is an array of size p cells.
- **Mem:** This is a matrix consisting of $(p-2) \times (n+1)$ cells.

Thus, total memory cells used by *HEOPTA* is equal to $m \times p + 2 \times p + (p-2) \times (n+1) \cong O(p \times (m+n))$. *End of Proof*.

Appendix D

HEPOPTA Details

In this appendix, the helper functions used in Chapter 5, “Bi-objective Optimization of Data-parallel Applications on Heterogeneous HPC Platforms for Performance and Energy Using Workload Partitioning”, will be explained, and the correctness and complexity of the proposed algorithm, *HEPOPTA*, will be proved.

D.1 Helper Routines Called in *HEPOPTA*

D.1.1 Function *ReadFunc*

The input parameters to the function READFUNC are F, which can be a discrete time ($t_i, i \in \{0, 1, \cdots, p-1\}$) or dynamic energy function ($e_i, i \in \{0, 1, \cdots, p-1\}$), and a problem size w (Algorithm 28). The function returns the execution time or the dynamic energy consumption of w from the function F. It returns 0 for zero problem sizes and -1 when there is no match for w in the function. This function uses functions sorted by problem size.

D.1.2 Function *SizeThresholdCalc*

The Algorithm 29 shows the pseudocode of the function SIZETHRESHOLDCALC which calculates the size threshold array, σ. First, It determines the size threshold of L_{p-1} by finding the greatest problem size in the energy function

Algorithm 28 Algorithm Reading the Execution Time or Energy Consumption of a Given Problem Size

```
1: function ReadFunc(F, w)
2:     if w = 0 then
3:         return 0
4:     end if
5:     if ∄(w, f_iw) ∈ F then
6:         return −1
7:     end if
8:     return f_iw
9: end function
```

E_{p-1} that its energy consumption is less than or equal to ε (Line 2). Then, it calculates σ_i, $i \in \{0, 1, , \cdots, p-2\}$ where σ_i is the summation of σ_{i+1} with the greatest work-size in energy function E_i that its consumed dynamic energy is less than or equal to ε (Lines 3-5).

Algorithm 29 Algorithm Determining Size Thresholds

```
1: function SizeThresholdCalc(p, E, ε, σ)
2:     σ_{p−1} ← max_{j=0}^{m−1} {x_{(p−1) j} | e_{(p−1) j} ≤ ε}
3:     for all i = p − 2; i ≥ 0; i−− do
4:         σ_i ← σ_{i+1} + max_{j=0}^{m−1} {x_ij | e_ij ≤ ε}
5:     end for
6:     return σ
7: end function
```

The function uses dynamic energy functions sorted in non-decreasing order of dynamic energy consumption.

D.1.3 Function *Cut*

The function Cut returns *TRUE* if the input workload n is greater than the input size threshold σ (Algorithm 30).

Algorithm 30 Algorithm Cutting Search Tree using the Size Threshold

```
1: function Cut(n, σ)
2:     if n > σ then
3:         return TRUE
4:     end if
5:     return FALSE
6: end function
```

D.1.4 Structure of matrix *PMem* in *HEPOPT*

We use *PMem*, a two-dimensional array, to memorize Pareto-front solutions for dynamic energy and performance which have been found at levels $\{L_1, \cdots, L_{p-2}\}$ in solution trees. Consider a given memory cell $PMem[i][n]$ which saves a Pareto-optimal solution which is found for a given workload n on processors $\{P_i, \cdots, P_{p-1}\}, i \in \{1, 2, \cdots, p-2\}$. The memory cell consists of a set where each element in this set is a tuple like $< eng, time, part, P\#, key >$, storing one Pareto-optimal solution.

The field eng stores the dynamic energy consumption of the Pareto-optimal solution on processors $\{P_i, \cdots, P_{p-1}\}$, $time$ is its parallel execution time on the processors, $part$ determines the problem size assigned to P_i by the solution, $P\#$ represents the number of active processors in the solution, and key is the dynamic energy consumption of a saved Pareto-optimal solution, provided it exists, for a node at the level L_{i+1} labelled by $n - part$ where this Pareto-optimal solution is the partial solution for the node n. Since dynamic energy consumptions are unique in Pareto-optimal sets, we use this parameter for pointing to partial solutions. In fact, key operates as a pointer to partial solutions.

Elements in Pareto-optimal sets are sorted in increasing order of dynamic energy consumption. If there exists no Pareto-optimal solution for the workload n on the level i, its corresponding memory cell, $PMem[i][n]$, will contain one tuple that its eng element is set to the constant value $\_NS$ (i.e. *No_Solution*).

D.1.5 Function *ReadParetoMem*

Algorithm 31 illustrates the function READPARETOMEM. Suppose we are going to retrieve the saved solutions for a given workload n on L_c. First, $PMem[c][n]$ is read which involves the saved solutions for n (Line 31). If $PMem[c][n]$ is empty, that is this node has not been visited yet, and the function returns *DUMMY* (Lines 3-5). In this case, *HEPOPTA_Kernel* will continue with expanding this node.

Since solutions in memory cells are sorted in the increasing order of the

dynamic energy consumption, we consider the energy consumption of the first element in each set as the best solution. According to the retrieved value for eng, the following cases might happen:

- **NOT_SOLUTION**: This case occurs when eng is equal to $\_NS$ (there is no solution for n on processor $\{P_c, \cdots, Pp-1\}$) or the consumed dynamic energy of the saved solution is greater than ε (Lines 6 and 8).
- **SOLUTION**: This case occurs if the retrieved eng is less than or equal to ε (Line 9).

Algorithm 31 Algorithm Retrieving Solution from Memory

```
1: function READPARETOMEM(n, c, ε, Mem)
2:     pSet ← PMem[c][n]
3:     if |pSet| = 0 then
4:         return DUMMY
5:     end if
6:     if pSet[0].eng = _NS ∨ pSet[0].eng > ε then
7:         return NOT_SOLUTION
8:     end if
9:     return SOLUTION
10: end function
```

D.1.6 Function *MakeParetoFinal*

Algorithm 32 illustrates the function MAKEPARETOFINAL which finalizes the input memory cell $pmem$. As explained in Chapter 5, each memory cell is finalized when its corresponding node along with its all children in the tree are completely explored. If a node is expanded for which there is no Pareto-optimal solution, the node labelled as $\_NS$ by inserting a tuple with the constant value $\_NS$ in field eng (Line 3). This means that there is no Pareto-front solution for this node.

Algorithm 32 Algorithm Finalizing Memory Cells

```
1: function MAKEPARETOFINAL(pmem)
2:     if |pmem| = 0 then
3:         pSet ← (_NS, 0, 0, 0, 0)
4:     end if
5: end function
```

D.1.7 Function *MergePartialParetoes*

For every non-leaf node, *HEPOPTA_Kernel* invokes the function MERGEPARTIALPARETOES to build its Pareto-optimal solutions, using the Pareto-optimal sets of its children which are named *partial solutions* for the node. The function then stores the new solutions in $PMem$. If there exist two workload distributions with equal dynamic energy consumption and execution time, MERGEPARTIALPARETOES selects the solution with the minimum number of active processors. The input variable c indicates a level in the tree, and $partsVec$ is a list including all problem sizes allocated to P_c where results in a solution. The algorithm starts with initializing $pSet$ which points to a memory cell storing Pareto-optimal solutions for a workload n on L_c (Lines 2-6). The set Ψ_{EP} will store final globally Pareto-optimal solutions for the root. The first For loop iterates all problem sizes in $partsVec$ and builds new feasible solutions by merging the problem sizes in $partsVec$ with their corresponding partial Pareto-optimal solutions (Lines 7-63). In each iteration, for a given problem size x, MERGEPARTIALPARETOES finds the partial Pareto-optimal solutions ($subPareto$) in $PMem$ (if $1 \leq c < p-2$) or builds it (if $c = p-2$) (Lines 8-16). The inner For loop scans all Pareto-optimal solutions in $subPareto$. It merges the problem size x, given to P_c, with Pareto-solutions in $subPareto$ for processors $\{P_{c+1}, \cdots, P_{p-1}\}$ (Lines 19-62). For each merged solution, $pSet$ is examined to verify whether there exists a Pareto-optimal solution in the set. If it is the case, $pSet$ is updated, and all non-Pareto-optimal solutions are eliminated. Therefore, for each newly merged solution $(eng_x, time_x, x, P\#_x, key)$, the following situations may happen:

1. $pSet$ is empty and the solution is inserted (Line 25).
2. There exists a solution in $pSet$ that its eng is equal to eng_x. In this case the saved solution is updated if either eng_x is less than eng or $P\#_x$ is less than $P\#$ (Lines 28-37).
3. The eng_x of the merged Pareto-optimal solution is greater than ones in the $pSet$. The solution is inserted in case its execution time $time_x$ is less than the last solution in $pSet$ (Lines 37-43).

4. The eng_x of the merged Pareto-optimal solution is less than ones in the $pSet$. The solution is inserted in $pSet$ after eliminating all non-Pareto-optimal solutions (Lines 43-49).

5. The eng_x of the merged solution is somewhere at the middle of $pSet$. In this case, the solution is inserted in $pSet$, and all non-Pareto-optimal solutions are removed (Lines 49-57).

It should be mentioned that the function *lower_bound* returns a pointer to the first element in the $pSet$ that its eng is greater than or equal to eng_x.

The algorithm prevents further iteration in $pSet$ if the execution time of the last-evaluated partial Pareto-optimal solution is less than or equal to the execution time of problem size x on P_c. In fact, the further scanning of the $pSet$ will not lead to a Pareto-optimal solution. This is because all Pareto-optimal solutions are sorted in increasing order of dynamic energy consumption, that consequently implies that the execution times are decreasing in each set. Thus, all solutions built using the following elements in the $pSet$ will have the same execution time as the execution time of the workload x on P_c but with greater energy consumption.

Finally, the function BUILDPARETOSOLS is called to obtain the workload distribution for each solution in Ψ_{EP} (Lines 64-66).

D.1.8 Function *BuildParetoSols*

As explained in the section D.1.7, the set Ψ_{EP} holds final globally Pareto-optimal solutions for dynamic energy and performance. Each element in Ψ_{EP}, which represents a Pareto-optimal solution, is a triple like $(eng, time, X)$ where eng determines the dynamic energy consumption of the solution, $time$ is its execution time, and $X = \{x_0, x_1, \cdots, x_{p-1}\}$ represents the workload distribution of the solution. The function BUILDPARETOSOLS determines the problem sizes given to the processors $\{P_1, \cdots, P_{p-1}\}$ by any solution in $PMem[0][0]$.

The algorithm 34 shows the pseudocode of BUILDPARETOSOLS. The function reads the problem sizes given to the processors $\{P_1, \cdots, P_{p-1}\}$ from

Algorithm 33 Algorithm Merging Partial-Pareto Solutions

```
1: function MERGEPARTIALPARETOES(n, p, c, E, T, partsVec, PMem, Ψ_EP)
2:     if c = 0 then
3:         pSet ← PMem[0][0]
4:     else
5:         pSet ← PMem[c][n]
6:     end if
7:     for all x ∈ partsVec do
8:         if c < p − 2 then
9:             subPareto ← PMem[c + 1][n − x]
10:        else
11:            x' ← n − x
12:            P#_x' ← (x = 0 ? 0 : 1)
13:            time_x' ← READFUNC(T_{P−1}, x')
14:            eng_x' ← READFUNC(E_{P−1}, x')
15:            subPareto ← (eng_x', time_x', x', P#_x', −)
16:        end if
17:        time_x ← READFUNC(T_c, x)
18:        P#_x ← (x = 0 ? 0 : 1)
19:        for all tup ∈ subPareto do
20:            eng_x ← tup.eng + READFUNC(E_c, x)
21:            time_x ← Max(tup.time, time_x)
22:            P#_x ← P#_x + P#_tup
23:            key ← tup.eng
24:            if |pSet| = 0 then
25:                pSet ← (eng_x, time_x, x, P#_x, key)
26:            else
27:                tup_l ← pSet.lower_bound(eng_x)
28:                if tup_l ≠ pSet.end() ∧ tup_l.eng = eng_x then
29:                    if tup_l.time > time_x then
30:                        tup_l ← (eng_x, time_x, x, P#_x, key)
31:                        for all r ∈ pSet | r.eng > eng_x ∧ r.time ≥ time_x do
32:                            pSet ← pSet − r
33:                        end for
34:                    else if tup_l.time = time_x ∧ P#_x < tup_l.P# then
35:                        tup_l ← (eng_x, time_x, x, P#_x, key)
36:                    end if
37:                else if tup_l = pSet.end() then
38:                    tup_l ← tup_l − 1
39:                    if tup_l.time > time_x then
40:                        pSet ∪ (eng_x, time_x, x, P#_x, key)
41:                    end if
42:                else if tup_l = pSet.begin() then
43:                    if time_x ≤ tup_l.time then
44:                        for all r ∈ pSet | r.eng > eng_x ∧ r.time ≥ time_x do
45:                            pSet ← pSet − r
46:                        end for
47:                    end if
48:                    pSet ∪ (eng_x, time_x, x, P#_x, key)
49:                else
50:                    tup_l ← tup_l − 1
51:                    if tup_l.time > time_x then
52:                        pSet ∪ (eng_x, time_x, x, P#_x, key)
53:                        for all r ∈ pSet | r.eng > eng_x ∧ r.time ≥ time_x do
54:                            pSet ← pSet − r
55:                        end for
56:                    end if
57:                end if
58:                if tup.time ≤ time_x then
59:                    break
60:                end if
61:            end if
62:        end for
63:    end for
64:    if c = 0 then
65:        BUILDPARETOSOLS(PMem, Ψ_EP)
66:    end if
67: end function
```

$PMem$. It uses the field key in each saved solution to find the corresponding partial solution and eventually the problem size give to P_{i+1}. Since energy consumptions are unique in any set, there is only one tuple that its dynamic energy consumption is equal to key in that set.

Algorithm 34 Algorithm Completing Workload Distribution for Ψ_{EP}

1: **function** BUILDPARETOSOLS($PMem, \Psi_{EP}$)
2: **for all** $tup \in PMem[0][0]$ **do**
3: $sumSize \leftarrow tup.part$
4: $X[0] \leftarrow tup.part$
5: $key_{cur} \leftarrow tup.key$
6: **for all** $i = 1; i \leq p-2; i{+}{+}$ **do**
7: $tup_{sub} \leftarrow \{t \in PMem[i][n - sumSize] \mid t.eng = key_{cur}\}$
8: $X[i] \leftarrow tup_{sub}.part$
9: $sumSize \leftarrow sumSize + tup_{sub}.part$
10: $key_{cur} \leftarrow tup_{sub}.key$
11: **end for**
12: $X[p-1] \leftarrow n - sumSize$
13: $\Psi_{EP} \leftarrow \Psi_{EP} \cup (tup.eng, tup.time, X)$
14: **end for**
15: **end function**

D.2 Correctness Proof of *HEPOPTA*

Proposition D.2.1. *The algorithm* HEPOPTA *always returns globally Pareto-optimal solutions.*

Proof. To obtain globally Pareto-optimal solutions for the dynamic energy and performance of a given workload n between p processors $\{P_0, \cdots, P_{p-1}\}$, we need all possible distributions for the workload. One approach is to employ the naive algorithm exploring full tree of solutions and build the globally Pareto-optimal set which suffers from exponential complexity. *HEPOPTA* enhances the naive approach using the specific operation *Cut* to just explore a small fraction of the full solution tree. Therefore, the correctness of *HEPOPTA* will be proved if we show that there exists no subtree ignored by the operation *Cut* while contains a Pareto-optimal solution.

Consider a given node which is labelled by n at a level $L_i, i \in \{0, 1, \ldots, p-2\}$ in a solution tree . The operation *Cut* removes the subtree growing from a node in case the workload of this node exceeds its corresponding size

threshold. Suppose the workload distribution $X = \{x_0, \cdots, x_{p-1}\}$ is eliminated from the search space by using *Cut* operation. Regarding the definition of size thresholds, the dynamic energy consumption of this workload ($E_D(X) = \sum_{i=0}^{p-1} E_i(x_i)$) is greater than ε ($\varepsilon < E_D(X)$). It should be mentioned that the execution time of this solution ($T_E(X) = \max_{i=0}^{p-1} T_i(x_i)$) will be greater than or equal with the optimal execution time for the workload n, t_{opt} ($t_{opt} \leq T_E(X)$). As explained in the main manuscript, ε is set to the dynamic energy consumption of the optimal distribution for execution time. Hence, there is a distribution like $X^* = \{x_0^* \cdots, x_{p-1}^*\}$ where its execution time and dynamic energy consumption are $T_E(X^*) = t_{opt}$ and $E_D(X^*) = \varepsilon$, respectively. Thus, we have $E_D(X^*) < E_D(X)$ and $T_E(X^*) \leq T_E(X)$, and according to the definition of Pareto-optimal sets, the solution X, which is removed by *Cut*, goes after the solutions X^* and cannot be a member of the Pareto-optimal set. *End of Proof*.

D.3 Complexity of *HEPOPTA*

Lemma D.3.1. *The maximum number of Pareto-optimal solutions for dynamic energy and performance on a heterogeneous platform including p discrete dynamic energy and p performance functions with a cardinality of m is equal to $m \times p$.*

Proof. We know that the execution time of a Pareto-optimal solution with the workload distribution $X = \{x_0, x_1, \cdots, x_{p-1}\}$ is equal to the execution time of an $x_i \in X$ where $T_i(x_i) = \max_{j=0}^{p-1} T_j(x_j)$, so that $T_i(x_i)$ represents the execution time of running x_i on P_i. In other words, the execution time of any distribution like X is equal to the execution time of one the problem sizes in X (i.e. $x_i \in X$) which has maximum execution time. Since we have p time functions with a cardinality of m, there exist up to $m \times p$ data points with different execution times. Therefore, one can conclude that the number of solutions with different execution times cannot go beyond $m \times p$.

On the other hand, regarding the definition of Pareto-optimality, we know that values for energy and performance are unique in any Pareto-optimal set

where there is not two solutions in one set where either their dynamic energy consumptions or their execution times are the same. Since there exist up to $m \times p$ distinct execution time, the cardinality of the Pareto-optimal set cannot exceed $m \times p$. *End of Proof*.

Lemma D.3.2. *The computational complexity of the function* MERGEPARTIAL-PARETOES *for building Pareto-optimal solutions of a given node in a solution tree for a heterogeneous platform including p discrete dynamic energy and p performance functions with a cardinality of m is equal to $O(m^2 \times p \times \log_2(m \times p))$.*

Proof. Consider a given node N at a level L_c of a solution tree. As explained in Chapter 5, the node has generally up to $m+1$ children. Regarding Lemma D.3.1, each child of N at L_{c+1} has up to $m \times (p-c-1)$ Pareto-optimal solutions. Consider Ψ_N, a data structure of the type *map*, storing the Pareto-optimal solutions of the node N. We know that the cardinality of Ψ_N does not exceed $m \times (p-c)$ Pareto-optimal solutions (Lemma D.3.1). To find the Pareto-optimal solutions of the node N, there are totally $(m+1) \times (m \times (p-c-1))$ merged solutions which should be examined one by one so that inserting a merged solution in Ψ_N has a complexity of $\log_2(m \times (p-c))$. Henceforth, the cost of processing and inserting all merged solutions is $(m+1) \times (m \times (p-c-1)) \times \log_2(m \times (p-c)) \cong O(m^2 \times p \times \log_2(m \times p))$. There exist around $(m+1) \times (m \times (p-c-1)) - m \times (p-c)$ non-Pareto-optimal solutions which are being eliminated from Ψ_N during the processing of the merged solutions. The elimination cost is totally $O(m^2 \times p)$. Therefore, the computational cost of merging all solutions for a given node in a search tree is equal to $O(m^2 \times p \times \log_2(m \times p))$. *End of Proof*

Lemma D.3.3. *The computational complexity of* HEPOPTA_Kernel *is $O(m^3 \times p^3 \times \log_2(m \times p))$.*

Proof. HEPOPTA_Kernel is an enhanced recursive algorithm, and therefore, its computational complexity can be related in terms of the number of its recursions. We will formulate the number of recursions using a trivial sample tree.

Let's consider a workload n running on 5 heterogeneous processors ($p = 5$). Suppose there exist five discrete performance, $T_i(x)$, and five discrete dynamic energy functions, $E_i(x)$ with a cardinality of 2 ($m = 2$) where $x = \{\Delta x, 2\Delta x\}$ and $i \in \{0, 1, \cdots, 4\}$. It should be noted that *HEPOPTA_Kernel* is able to deal with any granularity for workload sizes and considering the fix granularity size $\Delta x \in \mathbb{N}$ does not make the proof less general. Without loss of generality and for the sake of simplicity, we assume that execution time and dynamic energy consumption increase when problem size increases.

Figure D.1 shows the solution tree for finding the Pareto-optimal solutions for the workload n on the five processors. Let's n be greater than $8\Delta x$, the maximum possible workload which is subtracted from n in this example. In the figure, red nodes represent ones have been already expanded in the same level, and their solutions are retrieved from $PMem$. For the sake of simplicity, the operation *Cut* has not been employed. Thus, the practical time complexity is enormously less than the theoretical one.

According to the sample tree, the number of recursions (the number of nodes that their solutions do not retrieve form memory) in each level of solutions tree explored can be obtained using the Eq. C.1.

$$C\#(L) = \begin{cases} L \times m + 1 & 0 \leq L < p - 1 \\ C\#(p-2) \times (m+1) & L = p - 1 \end{cases} \tag{D.1}$$

where L represents the level number.

The expanded form of Eq. C.1 is shown in Eq. C.2.

$$C\#(L) = \begin{cases} L \times m + 1 & 0 \leq L < p - 1 \\ m^2 \times p - 2 \times m^2 + m \times p - m + 1 & L = p - 1 \end{cases} \tag{D.2}$$

That is, the total number of recursive calls is equal to $\sum_{L=0}^{p-1}(C\#(L))$ which is equal to $O(m \times p^2 + m^2 \times p)$.

In addition, the number of nodes in each level that their results are retrieved from $PMem$ is formulated in Eq. D.3.

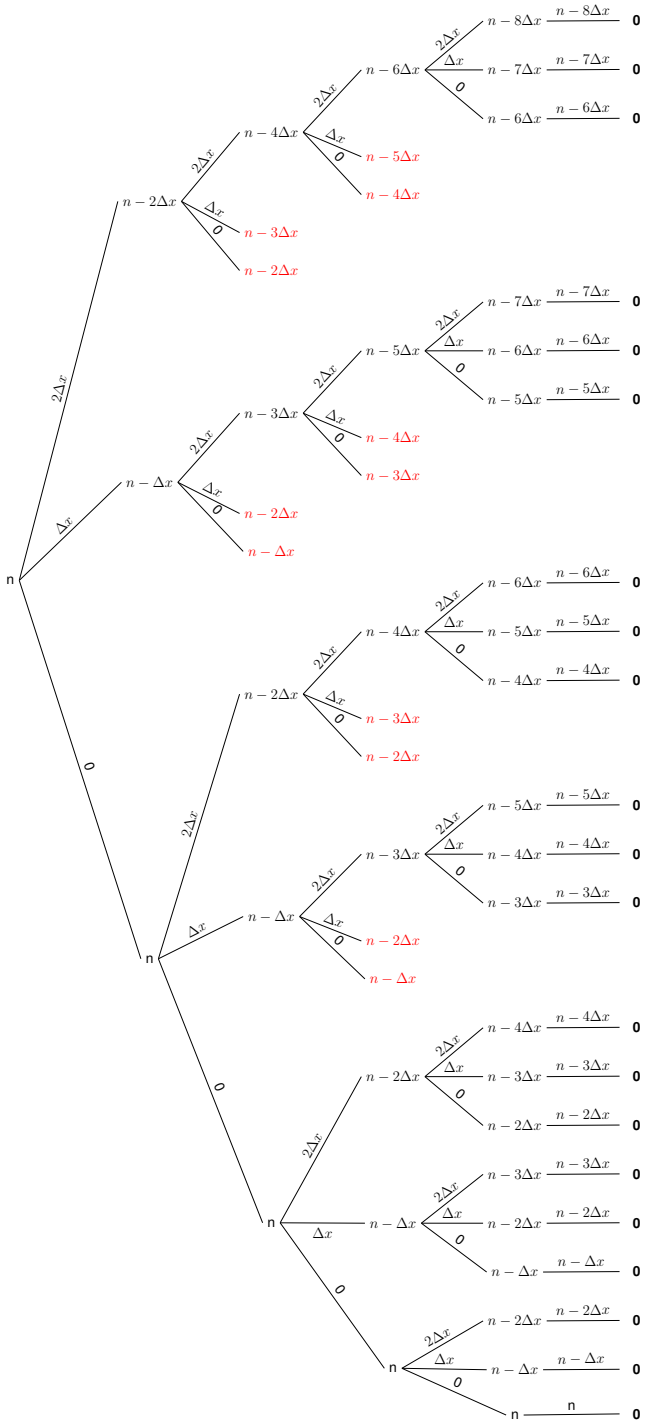

Figure D.1: The HEPOPTA *solution tree for executing a sample set of five profiles ($p = 5$), each contains 2 data points. The memorization technique is only considered to reduce the full search space of solutions.*

$$\begin{aligned} \text{Memory\#(L)} &= (C\#(L-1)-1) \times m \\ &= (m^2) \times (L-1), \quad 1 \leq L \leq p-2 \end{aligned} \tag{D.3}$$

Since $PMem$ saves the solutions which are found on levels 1 to $p-2$, the total number of nodes that their solutions are saved (nodes in red in the figure) is equal to $\sum_{L=1}^{p-2} Memory\#(L) = O(m^2 \times p^2)$. The complexity of READPARETOMEM is $O(1)$. Therefore, the computational cost for retrieving all solutions from $PMem$ is equal to $O(m^2 \times p^2)$.

The function MERGEPARTIALPARETOES is invoked after exploring all children of any node (nodes in black in Figure D.1) in levels $\{L_0, \cdots, L_{p-2}\}$. Regarding Lemma D.3.2 and Eq. D.1, the total cost of all MERGEPARTIALPARETOES calls is equal to $\sum_{L=0}^{p-2}(L \times m + 1) \times (m^2 \times p \times \log_2(m \times p)) = O(m^3 \times p^3 \times \log_2(m \times p))$.

The computational complexity of *HEPOPTA_Kernel* can be summarized as follows:

$$\begin{aligned} \text{Complexity(}\textit{HEPOPTA\_Kernel}\text{)} = &\text{O(recursive calls of } \textit{HEPOPTA\_Kernel}\text{)} + \\ &\text{O(}PMem\text{ solutions)} + \\ &\text{O(MERGEPARTIALPARETOES calls).} \end{aligned}$$

which equals:

$$\begin{aligned} \text{Complexity(}\textit{HEPOPTA\_Kernel}\text{)} = &O(m \times p^2 + m^2 \times p) + \\ &O(m^2 \times p^2) + \\ &O(m^3 \times p^3 \times \log_2(m \times p)) \\ &= O(m^3 \times p^3 \times \log_2(m \times p)). \end{aligned}$$

Proposition D.3.4. *The computational complexity of* HEPOPTA *is* $O(m^3 \times p^3 \times \log_2(m \times p))$.

Proof. HEPOPTA consists of following main steps:

- **Sorting:** There exist p discrete performance and p discrete dynamic energy profiles with a cardinality of m. The complexity to sort all of them is $O(p \times m \times \log_2 m)$.
- **Initializing energy threshold ε:** Obtaining energy threshold involves two steps: (i) invoking *HPOPTA* with a complexity of $O(m^3 \times p^3)$ followed by (ii) calculating the energy threshold ε with a complexity of $O(p)$. Therefore, the complexity of this step is equal to $O(m^3 \times p^3)$.
- **Finding size thresholds:** To find the size threshold a given level L_i, $i \in [0, p-1]$, all data points, existing in $e_i(x)$ with dynamic energy consumptions greater than ε should be examined in a complexity of $O(m)$. Therefore, finding p size thresholds has a complexity of $O(p \times m)$.
- **Memory initialization:** In this step, all $(n+1) \times (p-2)$ cells of $PMem$ are initialized with a complexity of $O(n \times p)$.
- **Kernel invocation:** According to Lemma D.3.3, the complexity of *HEPOPTA_Kernel* is $O(m^3 \times p^3 \times \log_2(m \times p))$.

Thus, the computational complexity of *HEPOPTA* is equal to the summation of all these steps, which is equal to $O(m^3 \times p^3 \times \log_2(m \times p))$. *End of Proof*.

Proposition D.3.5. *The total memory consumption of* HEPOPTA *is* $O(n \times m \times p^2)$.

Proof. HEPOPTA uses memory to store following information:

- **energy functions:** There are p discrete energy functions with cardinality of m. We store both size-sorted (sorted by problem size) and energy-sorted (sorted by the amount of dynamic energy consumption) functions. These function are stored in $2 \times p \times m$.
- **time functions:** There are p discrete time functions with cardinality of m. We store both size-sorted (sorted by problem size) and time-sorted (sorted by execution time) functions. These function are stored in $2 \times p \times m$.

- Ψ_{EP}**:** Regarding Lemma D.3.1, the maximum number of Pareto-optimal solutions are $m \times p$. Since the workload distribution of each solution, including p elements, is stored in Ψ_{EP}, the maximum size of the set Ψ_{EP} is $O(m \times p^2)$.

- **PMem:** This is a matrix consisting of $(p-2) \times (n+1)$ cells. Each cell stores up to $m \times p$ Pareto-optimal solutions (Lemma D.3.1). Therefore, the memory usage of $PMem$ is equal to $O(n \times m \times p^2)$.

- **Memory consumption of *HPOPTA*:** The memory usage of *HPOPTA* is $O(p \times (m+n))$.

- X_{cur}: This is an array of p elements to store the problem sizes assigned to each processor by the current solution.

- **partsVec**: This is a vector of size $O(m)$ storing the problem sizes given to a processor where results in a solution. There exists $p-1$ partsVecs, one per level. That it total consumed memory is equal to $O(m \times p)$.

Thus, an upper bound for the total memory usage of *HEPOPTA* is equal to $O(n \times m \times p^2)$. *End of Proof.*

Appendix E

HTPOPTA Details

In this chapter, We will prove how to calculate globally Pareto-optimal solutions for total energy and performance using globally Pareto-optimal solutions for dynamic energy and performance. Then, the correctness and computational complexity proofs of *HTPOPTA* will be explained.

E.1 Definition of Pareto-optimal Solutions for Dynamic Energy and Execution Time

Suppose there exists a given workload n executing on p processors using a workload distribution $X^* = \{x_0^*, x_1^*, \cdots, x_{p-1}^*\}$ where $\sum_{i=0}^{p-1} x_i^* = n$, $E_D(X^*) = \sum_{i=0}^{p-1} e_i(x_i^*)$ is the dynamic energy consumption of the distribution, and $T_E(X^*) = \max_{i=0}^{p-1} t_i(x_i^*)$ represents its execution time.

Suppose S represents the feasible set of distributions. According to the definition of Pareto-optimality, the distribution X^* would be a Pareto-optimal solution if its dynamic energy consumption ($E_D(X^*)$) and execution time ($T_E(X^*)$) satisfy Eq. E.1. In this equation, $X = \{x_0, x_1, \cdots, x_{p-1}\} \in S$ represents any workload distribution for n.

$$\begin{aligned} &\nexists X \in S \mid E_D(X) \leq E_D(X^*) \wedge T_E(X) < T_E(X^*) \\ &AND \\ &\nexists X \in S \mid T_E(X) \leq T_E(X^*) \wedge E_D(X) < E_D(X^*) \end{aligned} \tag{E.1}$$

Equation E.1 means that there does not exist any objective vector $(E_D(X), T_E(X))$ for which all the objective vector values are less than Pareto-optimal vector $(E_D(X^*), T_E(x^*))$. In fact, there is no other solution which dominates X^*.

Lemma E.1.1. *For each non-Pareto-optimal workload distribution X with the objective vector $(E_D(X), T_E(X))$, there is at least one Pareto-optimal solution X^* where either $E_D(X^*) \leq E_D(X)$ and $T_E(X^*) < T_E(X)$ or $T_E(X^*) \leq T_E(X)$ and $E_D(X^*) < E_D(X)$.*

Proof. Regarding the Eq. E.1, for each non-Pareto-optimal solution X exists at least one solution Y in the objective space where $E_D(Y) \leq E_D(X) \wedge T_E(Y) < T_E(X)$ or $T_E(Y) \leq T_E(X) \wedge E_D(Y) < E_D(X)$. The point X is called *dominant point*. If the dominant point Y is a Pareto-optimal solution, the correctness of the lemma is proven. But if not so, there exists another dominant point so that dominates Y. The process of finding dominant points can be recursively repeated. The recursion will eventually terminate because the two objectives execution time and dynamic energy consumption are *finite positive* parameters and their values gradually decrease during this recursive process. That is, the recursive process finally reaches a given solution X^* which cannot be dominated by any other solution. According to the definition of Pareto-optimal solutions, the solution X^* should be a member of the Pareto-optimal set for dynamic energy and execution time (Ψ_{EP}). Therefore, the correctness of the lemma E.1.1 is proven. *End of Proof*.

E.2 Pareto-front Solutions for Total Energy and Execution Time

In this section, we will prove how to build Pareto-optimal solutions for total energy and execution time (Ψ_{TP}) by using Pareto-optimal solutions for dynamic energy and execution time (Ψ_{EP}).

Proposition E.2.1. *There is no workload distribution X such that $X \notin \Psi_{EP}$ but $X \in \Psi_{TP}$.*

Proof. Referring to Lemma E.1.1, if a solution X is not in Ψ_{EP} then there exists a Pareto-optimal solution X^* such that either $E_D(X^*) \leq E_D(X)$ and $T_E(X^*) < T_E(X)$ or $T_E(X^*) \leq T_E(X)$ and $E_D(X^*) < E_D(X)$. Since total energy is a function of dynamic energy and execution time, it can be deducted that $E_T(X^*) < E_T(X)$. Since $T_E(X^*) < T_E(X)$ and either $E_T(X^*) < E_T(X)$ or $E_T(X^*) \leq E_T(X)$, the objective vector $(E_T(X^*), T_E(X^*))$ dominates $(E_T(X), T_E(X))$. Therefore, it can be concluded that feasible solutions which are not a remember of Ψ_{EP} cannot be a member of Ψ_{TP}. *End of Proof.*

Proposition E.2.2. *If X_{opt} is a workload distribution minimising total energy consumption, its corresponding objective vector, $(E_D(X_{opt}), T_E(X_{opt}))$, is necessarily a member of Pareto-optimal set for dynamic energy and performance.*

Proof. We categorize all points in the feasible set of distributions into two distinct groups: (i) solutions existing in the Pareto-optimal set Ψ_{EP}, and (ii) solutions are not Pareto-optimal. Regarding Lemma E.1.1, for each non-Pareto-optimal solution X, there is at least one solution X^* in Pareto-optimal set dominating X. Since total energy is a function of dynamic energy and execution time, it can be concluded that

$$\forall x \notin \Psi_{EP}, \exists X^* \in \Psi_{EP}$$

$$where$$

$$E_T(X^*) < E_T(X)$$

That is, the solution which minimizes total energy must be a member of Pareto-optimal set. *End of Proof*.

E.3 Complexity of *HTPOPTA*

Proposition E.3.1. *The computational complexity of* HTPOPTA *is* $O(m^3 \times p^3 \times \log_2(m \times p))$.

Proof. To find globally Pareto-optimal solutions for total energy and performance, *HTPOPTA*, first, invokes *HEPOPTA* for obtaining Ψ_{EP}, with a complexity of $O(m^3 \times p^3 \times \log_2(m \times p))$.

It then calculates the total energy consumption of each solution up to $m \times p$ number of solutions in Ψ_{EP} and inserts the new solutions into Ψ_{TP} or updates existing ones. Ψ_{TP} is defined as a data structure of the type *map* to store Pareto-optimal solutions for total energy and performance. In the case of existing two solutions with equal total energy consumption and execution times, the solution with less active processors (processors with non-zero workloads) is chosen by *HTPOPTA*. Inserting a solution in Ψ_{TP} has a logarithmic computational complexity, and determining solutions with less active processors has a complexity of $O(p)$. Therefore, the cost of inserting and updating up to $m \times p$ solutions in Ψ_{TP} is $(m \times p) \times (\log_2(m \times p) + p) \cong O(m \times p^2)$. In addition, the computational complexity for eliminating all non-Pareto-optimal solutions from Ψ_{TP} is $O(m \times p)$.

Therefore, the total computational cost to calculate Ψ_{TP} is equal to $O(m^3 \times p^3 \times \log_2(m \times p))$. *End of Proof*.

Proposition E.3.2. *The total memory consumption of* HTPOPTA *is* $O(n \times m \times p^2)$.

Proof. HTPOPTA uses memory to store following information:

- **energy functions:** There are p discrete energy functions with a cardinality of m. We store both size-sorted (sorted by problem size) and energy-sorted (sorted by the amount of dynamic energy consumption) functions. These function are stored in $2 \times p \times m$.

- **time functions:** There are p discrete time functions with a cardinality of m. We store both size-sorted (sorted by problem size) and time-sorted (sorted by execution time) functions. These function are stored in $2 \times p \times m$.

- **Required memory by *HEPOPTA:*** As proved in Appendix D, the total memory consumption of *HEPOPTA* is $O(n \times m \times p^2)$.

- Ψ_{TP}**:** As proved in Appendix D (Lemma D.3.1), the maximum number of Pareto-optimal solutions is equal to $m \times p$. Since the workload distribution of each solution, involving p elements, is stored in Ψ_{TP}, the maximum size of Ψ_{TP} is $O(m \times p^2)$.

Thus, total memory usage of *HTPOPTA* is equal to $O(n \times m \times p^2)$. *End of Proof.*

Appendix F

Interfaces to Proposed Tools

In this chapter, we describe the interfaces to the routines provided by *HPOPTA*, a tool for performance optimization, *HEOPTA*, a tool for dynamic energy optimization, and *HEPOPTA*, a tool for bi-objective optimization for dynamic energy and performance, and total energy and performance for modern heterogeneous HPC platforms.

F.1 Interface to *HPOPTA*

HCL_hpopta

The implementation of the *HPOPTA* algorithm for heterogeneous clusters. The goal of the algorithm is to determine optimal workload distribution minimizing the execution time of a computation in its parallel execution.

Synopsis:

```
int
HCL_hpopta(
    const int verbosity,
    const unsigned int n,
    const unsigned int p,
    const unsigned int* npoints,
    const unsigned int* psizes,
```

```
    const double* etimes,
    unsigned int* xOpt,
    double* eOpt
);
```

Parameters:

- **verbosity** — Level of verbosity.
- **n** — The workload size.
- **p** — Number of processors.
- **npoints** — The number of points in each discrete time function in an array of size p.
- **psizes** — A list of problem sizes in p discrete time functions. Values for the first function are followed by those of the second one and so on.
- **etimes** — A list of execution times in p discrete time functions. Values for the first function are followed by those of the second one and so on.
- **xOpt** — The optimal workload distribution output in an array of size p.
- **eOpt** — The optimal execution time.

Usage:

```
/*appname.cpp using the hcl_hpopta interface to find optimal
 workload distribuion for performance.*/
#include <iostream>
#include "hcl_hpopta.h"

using namespace std;

int main(int argc, char** argv)
{
```

```
    unsigned int npoints[2] = {4,3};
    unsigned int psizes[7] = {1,2,3,4,1,2,3};
    double etimes[7] = {10,30,20,25,15,25,35};
    unsigned int xOpt[2];
    double eOpt;

    int rc = hcl_hpopta(1,
                   4,
                   2,
                   npoints,
                   psizes,
                   etimes,
                   xOpt,
                   &eOpt
    );

    if (rc == 0)
    {
       return 0;
    }
    else
    {
       //Error has occured
       return -1;
    }
}
```

Compilation: Use the command below to compile an application named *appname.cpp*:

```
$ g++ -I <path to hcl_hpopta.h> <appname.cpp> -L <path to
the library hpopt> -l hpopt
```

Return values: 0 on success, -1 in case of failure.

F.2 Interface to *HEOPTA*

HCL_heopta

The implementation of the *HEOPTA* algorithm for heterogeneous clusters. The goal of the algorithm is to determine optimal workload distribution minimizing the dynamic energy consumption of a computation in its parallel execution.

Synopsis:

```
int
HCL_heopta(
    const int verbosity,
    const unsigned int n,
    const unsigned int p,
    const unsigned int* npoints,
    const unsigned int* psizes,
    const double* energies,
    unsigned int* xOpt,
    double* eOpt
);
```

Parameters:

- **verbosity** — Level of verbosity.
- **n** — The workload size.
- **p** — Number of processors.

- **npoints** — The number of points in each discrete energy function in an array of size p.
- **psizes** — A list of problem sizes in p discrete energy functions. Values for the first function are followed by those of the second one and so on.
- **energies** — A list of dynamic energy consumptions in p discrete energy functions. Values for the first function are followed by those of the second one and so on.
- **xOpt** — The optimal workload distribution output in an array of size p.
- **eOpt** — The optimal dynamic energy consumption.

Usage:

```
/*appname.cpp using the hcl_heopta interface to find optimal
 workload distribuion for dynamic energy.*/
#include <iostream>
#include "hcl_heopta.h"

using namespace std;

int main(int argc, char** argv)
{
   unsigned int npoints[2] = {4,3};
   unsigned int psizes[7] = {1,2,3,4,1,2,3};
   double energies[7] = {10,30,20,25,15,25,35};
   unsigned int xOpt[2];
   double eOpt;

   int rc = hcl_heopta(1,
                  4,
                  2,
                  npoints,
                  psizes,
```

```
                    energies,
                    xOpt,
                    &eOpt
    );

    if (rc == 0)
    {
       return 0;
    }
    else
    {
       //Error has occured
       return -1;
    }
}
```

Compilation: Use the command below to compile an application named $appname.cpp$:

```
$ g++ -I <path to hcl_heopta.h> <appname.cpp> -L <path to
the library heopt> -L <path to the library hpopt> -l heopt
-l hpopt
```

Return values: 0 on success, -1 in case of failure.

F.3 Interface to *HEPOPTA* and *HTPOPTA*

HCL_hepopta

The implementation of the *HEPOPTA* and *HTPOPTA* for heterogeneous clusters. The algorithm aims to find Pareto-optimal solutions for dynamic en-

ergy and performance. It also calculates Pareto-optimal solutions for total energy and performance in case the input parameter bp, determining the amount of base power, is set to a positive value.

Synopsis:

```
int
HCL_hepopta(
   const int verbosity,
   const unsigned int n,
   const unsigned int p,
   const double bp,
   const unsigned int *npoints,
   const unsigned int* psizes,
   const double* etimes,
   const double* energies,
   unsigned int* nDEParetoSolutions,
   double** DEparetoSols,
   unsigned int* nTEParetoSolutions,
   double** TEparetoSols
);
```

Parameters:

- **verbosity** — Level of verbosity.
- **n** — The workload size.
- **p** — Number of processors.
- **bp** — The amount of base power consumption of the platform. The algorithm only calculates Pareto-optimal solutions for dynamic energy and performance in case bp is set to 0.
- **npoints** — The number of points in each discrete energy function in an array of size p.

- **psizes** — A list of problem sizes in p discrete energy functions. Values for the first function are followed by those of the second one and so on.
- **etimes** — A list of execution times in p discrete time functions. Values for the first function are followed by those of the second one and so on.
- **energies** — A list of dynamic energy consumptions in p discrete energy functions. Values for the first function are followed by those of the second one and so on.
- **nDEParetoSolutions** — The number of Pareto-optimal solutions for performance and dynamic energy.
- **DEparetoSols** — A list of Pareto-optimal solutions for performance and dynamic energy including $nDEParetoSolutions$ solutions.
- **nTEParetoSolutions** — The number of Pareto-optimal solutions for performance and total energy.
- **TEparetoSols** — A list of Pareto-optimal solutions for performance and total energy including $nTEParetoSolutions$ solutions.

Usage:

```
/*appname.cpp using the hcl_hepopta interface to find Pareto
-optimal solutions for performance and dynamic energy and
performance and total energy.*/
#include <iostream>
#include "hcl_hepopta.h"

using namespace std;

int main(int argc, char** argv)
{
   unsigned int npoints[2] = {4,3};
   unsigned int psizes[7] = {1,2,3,4,1,2,3};
   double etimes[7] = {10,30,20,25,15,25,35};
```

```
    double energies[7] = {10,30,20,25,15,25,35};
    unsigned int nDEParetoSolutions;
    double** DEparetoSols;
    unsigned int nTEParetoSolutions;
    double** TEparetoSols;

    int rc = hcl_hepopta(1,
                  4,
                  2,
                  100,
                  npoints,
                  psizes,
                  etimes,
                  energies,
                  &nDEParetoSolutions,
                  DEparetoSols,
                  &nTEParetoSolutions,
                  TEparetoSols
    );

    if (rc == 0)
    {
       return 0;
    }
    else
    {
       //Error has occured
       return -1;
    }
}
```

Compilation: Use the command below to compile an application named

appname.cpp:

```
$ g++ -I <path to hcl_hepopta.h> <appname.cpp> -L <path to
the library hepopt> -L <path to the library heopt> -L <path
 to the library hpopt> -l hepopt -l heopt -l hpopt
```

Return values: 0 on success, -1 in case of failure.

Appendix G

List of Abbreviations

G.1 Acronyms

The following describes the significance of various acronyms and terms used throughout this thesis. The page on which each one is defined or used is also given.

Acronyms

BLAS Basic Linear Algebra Subprograms. 49, 161, 167

BOPPE Bi-objective Optimization Problem for Performance and dynamic Energy. 13, 14, 130, 157

CPM Constant Performance Model. 4, 29, 78

DAG Directed Acyclic Graph. 21

DGEMM Double-precision General Matrix Multiplication. 5

DVFS Dynamic Voltage and Frequency Scaling. 8, 33, 129

EULAG Eulerian/semi-Lagrangian fluid solver. 4

FFT Fast Fourier Transform. 5, 51

FLOPS Floating Point Operations Per Second. 79, 92

FPGA Field Programmable Gate Array. 18, 20, 51, 159

FPM Functional Performance Model. 4, 78

GPU Graphics Processing Unit. 18, 51, 94, 159

HPC High Performance Computing. 1, 18, 92, 179

INLP Integer Non-Linear Programming. 58, 102, 112, 113, 180

LLC Last Level Cache. 1, 51, 179

MIC Many Integrated Core. 2

MIMD Multiple Instruction, Multiple Data. 18

MPDATA Multidimensional Positive Definite Advection Transport Algorithm. 4, 71

NIC Network Interface Card. 99

NUMA Non-Uniform Memory Access. 1, 19, 51, 179

PCA Principal Component Analysis. 33

PMC Performance Monitoring Counter. 25, 97

QPI Quick Path Interconnect. 1, 55, 179

SSD Solid State Drive. 99

UMA Uniform Memory Access. 19

Xeon Phi Intel Xeon Phi. 18, 51, 94, 159