Poznań University of Technology
Faculty of Computing and Telecommunications

PhD Thesis

Energy-aware resource management for stencil computations in High Performance Computing

Miłosz Ciżnicki

Supervisor: prof. zw. dr hab. inż. Jan Węglarz
Assistant supervisor: dr hab. inż. Krzysztof Kurowski

Poznań, 2023

Abstract

Parallel applications need to change even now by redesigning algorithms and data structures, respectively, to take advantage of the recent improvements in the energy efficiency of heterogeneous computing hardware, including multicore processors and GPU accelerators. Over the next few years, one of the biggest challenges for exascale computing will be the ability of parallel applications to fully exploit data locality, which will, in turn, be required to achieve the expected performance and energy efficiency. The future highly parallel applications will have to deal with deep memory hierarchies taking into account the energy cost in moving data off-chip. Therefore, they will have to apply new coordinated scheduling approaches to balance energy-aware resource utilization and minimise work starvation during runtime.

Stencil computations as a relevant class of applications occur in many HPC codes on block-structured grids for modelling various physical phenomena, e.g. computational fluid dynamics, geometric modelling, solving partial differential equations or image and video processing. As computing time and memory usage grow linearly with the number of array elements in stencil computations, our research targets highly parallel implementations of stencil codes together with task scheduling and optimisation techniques taking into consideration the energy cost and data locality. During our experimental studies we have proved that recent changes introduced in heterogeneous computing hardware resulted in different performance and energy characteristics that are critical for highly efficient scalable stencil computations.

To the best of our knowledge, none of the previous research considered an energy-aware distribution of the stencil workload on heterogeneous computing resources with the time constraint. Moreover, none of them tried to minimise the energy consumption of intra-node and inter-node communications that significantly influence energy savings.

In this thesis a topology-aware scheduling model is formulated and presented for efficient stencil computations executions taking into account different intra-node and inter-node communication links among heterogeneous processors. An innovative analytical and methodological approach is described for modelling and predicting energy usage and execution time for reference stencil patterns on single and multi-

node heterogeneous HPC architectures. New heuristic algorithms for simultaneously minimising energy usage and runtime of small and large scale stencil computations are presented. The proposed energy-aware resource management method considers complex hardware HPC architectures, including different and heterogeneous computing setups and communication topologies. A new Tabu Search algorithm is described for solving the problem and results of computational experiments comparing its performance vs simpler heuristics are presented. A flexible and generic scheduling model and heuristics algorithms are developed that can be easily extended or adapted in existing stencil application frameworks.

Streszczenie

Obecnie istniejące aplikacje równoległe muszą ewoluować poprzez przeprojektowanie algorytmów i struktur danych, aby wykorzystać ostatnie udoskonalenia w wydajności energetycznej heterogenicznych architektur komputerowych, włączając wielordzeniowe procesory i akceleratory GPU. W kolejnych latach jednym z największych wyzwań dla obliczeń exascale będzie osiągniecie zdolności aplikacji równoległych do pełnego wykorzystania lokalności danych, która będzie wymagana, aby uzyskać oczekiwaną wydajność obliczeniową i efektywność energetyczną. Opracowywane w przyszłości aplikacje równoległe będą musiały zmierzyć się z wielopoziomowymi hierarchiami pamięci, uwzględniając koszt energetyczny niezbędny do komunikowania się i przenoszenia danych poza układ scalony. W związku z tym będą musiały być opracowane i zastosowane nowe skoordynowane metody szeregowania, aby zrównoważyć zużycie energii zasobów oraz zminimalizować zagłodzenie procesów podczas ich wykonywania.

Obliczenia stencilowe stanowią ważną klasą aplikacji, które pojawiają się w wielu kodach HPC wykonywanych na blokowych siatkach strukturalnych używanych do modelowania różnych zjawisk fizycznych, przykładowo obliczeniowa mechanika płynów, geometria obliczeniowa, rozwiązywanie równań różniczkowych cząstkowych lub modyfikacja zdjęć i wideo. Ze względu na to, że czas przetwarzania i zużycie pamięci rośnie liniowo wraz z liczbą elementów siatki, nasze badania skupiają się na równoległych implementacjach obliczeń stencilowych razem z szeregowaniem zadań i technikami optymalizacji biorącymi pod uwagę koszt energetyczny i lokalność danych. Podczas naszych badań udowodniliśmy, że ostatnie zmiany wprowadzone w heterogenicznych architekturach komputerowych doprowadziły do tego, że różne cechy wydajnościowe i energetyczne są krytyczne dla efektywnych i skalowalnych obliczeń stencilowych.

Według stanu wiedzy, żadne z dotychczasowych badań nie uwzględniało tak szczegółowej analizy zużycia energii przy dystrybucji obliczeń stencilowych na heterogenicznych zasobach z ograniczeniami czasu wykonania. Dla wybranych referencyjnych obliczeń stencilowych dotychczasowe badania nie próbowały minimalizować zużycia energii na komunikacje wewnątrz węzła, jak również jak i między węzłami

obliczeń w odniesieniu do różnych topologii ich połączeń, co jak zostało wykazane w rozprawie znacząco może wpływać na oszczędność energii w szczególności w przypadku większych obliczeń.

W niniejszej rozprawie sformułowano i zaprezentowano model szeregowania dla efektywnych obliczeń stencilowych. Topologia uwzględnia różne połączenia komunikacyjne w ramach węzła, jak i pomiędzy węzłami dla heterogenicznych procesorów. Opisano innowacyjne, analityczne oraz metodologiczne podejście modelowania i przewidywania zużycia energii i czasu wykonania dla referencyjnych obliczeń stencilowych na pojedynczych, jak i wielu węzłach dla heterogenicznych architektur HPC. Przebadano i zaprezentowano nowe heurystyczne algorytmy dla równoczesnej minimalizacji zużycia energii i czasu wykonania dla małych oraz dużych zadań obliczeń stencilowych. Zaproponowano metodę zarządzania zadaniami, uwzględniającą zużycie energii, która bierze pod uwagę złożone sprzętowe architektury HPC, wliczając w to różne konfiguracje sprzętowe oraz topologie sieciowe na bazie rzeczywistych największych instalacji superkomputerowych. Opisano nowy algorytm przeszukiwania tabu w celu rozwiązania problemu i przedstawiono wyniki eksperymentów obliczeniowych, uwzględniając jego wydajność w porównaniu do prostych heurystyk. Opracowano elastyczny i generyczny model szeregowania oraz algorytmy heurystyczne, które mogą być łatwo rozszerzane albo zaadaptowane w istniejących platformach programistycznych dla obliczeń stencilowych.

Contents

Abstract **1**

Streszczenie **3**

1 Introduction **9**

1.1 Motivation for energy-aware stencil computations . . . 9
1.2 Goals and scope of the thesis . . . 10
1.3 Contributions presented in this thesis . . . 11
1.4 Structure of the dissertation . . . 12

2 High Performance Computing **13**

2.1 Architecture overview . . . 13
2.1.1 Processing Units . . . 15
2.1.2 Intra-node communication . . . 16
2.1.3 Inter-node communication . . . 19
2.1.4 Fat-tree network topology . . . 19
2.1.5 Torus network topology . . . 21
2.1.6 Dragonfly network topology . . . 23
2.2 Parallel application programming and execution environments for stencil computations . . . 25
2.2.1 Multi-node parallel programming environments . . . 26
2.2.2 Domain Specific Languages for stencil computations . . . 29
2.3 Selected tools . . . 30
2.3.1 Performance measurement . . . 30
2.3.2 Energy measurement . . . 30

3 Stencil computations **33**

3.1 Definition . . . 33
3.2 Performance optimisation methods . . . 35
3.2.1 Single processing unit . . . 37
3.2.2 Multiple processing units . . . 37

3.3 Performance models . . . 39
3.3.1 Roofline model . . . 40
3.3.2 Cache Aware Roofline model . . . 42
3.4 Methods to optimise energy consumption . . . 44
3.5 Energy models . . . 45

4 Basic notions in the theory of algorithms and computational complexity 49
4.1 Computational complexity . . . 49
4.2 Linear programming . . . 52
4.3 Graph theory . . . 53
4.3.1 Basic definitions . . . 53
4.3.2 Multiplicity . . . 54
4.3.3 Adjacency and incidence . . . 54
4.3.4 Maximum degree and maximum multiplicity . . . 54
4.3.5 Edge colouring . . . 54

5 Energy-aware resource management of stencil computations 57
5.1 Designation of the parameters . . . 57
5.2 Problem formulation . . . 63
5.3 Performance and energy model . . . 65
5.4 Time measurement . . . 66
5.4.1 Code analysis . . . 66
5.4.2 Hardware performance counters . . . 69
5.4.3 Instrumentation . . . 70
5.5 Energy measurement . . . 71

6 Solution methods 73
6.1 Exact method . . . 73
6.2 Heuristic algorithms . . . 76
6.2.1 Load Balancing . . . 77
6.2.2 Degree Minimisation . . . 77
6.2.3 Multicut Minimisation . . . 78
6.2.4 Neighbours Accumulation . . . 79
6.3 Computational experiments . . . 80
6.3.1 Simulation setup . . . 80
6.3.2 Simulation results . . . 81
6.3.3 Verification of energy model . . . 88

7 Task Movement algorithm 91
7.1 Single-node setup and new requirements . . . 91
7.2 Algorithm for multiple-node setup . . . 92
7.3 Computational experiments . . . 95
7.3.1 Simulation setup . . . 95

7.3.2 Simulation results 98

8 Conclusions 103

Bibliography 105

A Software and data repository 117

Chapter 1
Introduction

1.1 Motivation for energy-aware stencil computations

The performance of high-end supercomputers reached the exascale through the advent of core counts in billions. However, in the upcoming exascale computing era, it is crucial to focus not only on the performance but also the scalability of fine-grained parallel applications, data locality and energy-aware scheduling within the parallel code. Parallel applications need to change even now by redesigning algorithms and data structures, respectively, to take advantage of the recent improvements in the energy efficiency of heterogeneous computing hardware, including multicore processors and GPU accelerators. Over the next few years, one of the biggest challenges for exascale will be the ability of parallel applications to fully exploit locality, which will, in turn, be required to achieve the expected performance and energy efficiency. The future highly parallel applications will have to deal with deep memory hierarchies taking into account the energy cost in moving data off-chip. Therefore, they will have to apply new coordinated scheduling approaches to balance energy-aware resource utilization and minimise work starvation during runtime. As new constraints and limits on memory bandwidth and energy will play a key role in High Performance Computing (HPC) in the future, more sophisticated and dynamic scheduling techniques will be needed and applied within the parallel code.

Stencil computations as a relevant class of applications occur in many HPC codes on block-structured grids for modelling various physical phenomena, e.g. computational fluid dynamics, geometric modelling, solving partial differential equations or image and video processing [14, 22, 23, 85, 24]. As computing time and memory usage grow linearly with the number of array elements in stencil computations, our research targets highly parallel implementations of stencil codes together with task scheduling and optimisation techniques taking into consideration the energy cost and data locality [58, 68, 19, 11, 96]. During our experimental studies we have proved that recent changes introduced in heterogeneous computing hardware resulted in

different performance and energy characteristics that are critical for highly efficient and scalable stencil computations [25]. As shown in [73, 99], the overall performance of stencil computations is memory-bound. One should note that many existing HPC architectures mainly focus on floating-point performance [91]. However, only a partial and limited usage of the floating-point units in a given computing architecture is possible today and may reduce energy costs without performance degradation. Moreover, many latest improvements introduced in dynamic power management policies at the hardware level, e.g., dynamic voltage and frequency scaling (DVFS) or even switching off an entire unit block of a chip (clock gating) can lead to a significant reduction in the energy required for memory-bound workloads. Advanced dynamic power management policies give new opportunities for scheduling tasks within the fine-grained parallel code as users are able to control the utilisation of various functional units in heterogeneous computing hardware, e.g., turn on and off dynamically individual cores, change the frequency of small processing and communication units on-demand or even put portions of cache memory at specific sleep states during runtime.

All aforementioned problems and challenges regarding energy-aware stencil computations are addressed in this thesis.

1.2 Goals and scope of the thesis

The main thesis of this research is the following:

It is possible to develop and verify models experimentally for efficient distribution of the stencil workload on many heterogeneous processors and connected nodes. It can be demonstrated in practice how to efficiently explore the relationship between task scheduling algorithms and energy constraints based on a relevant and important class of stencil computations in supercomputing systems with different communication and network topologies.

To achieve this objective, the analysis and development of energy and performance models are required for the state-of-the-art mutli- and many-core supercomputing architectures. First of all, we need to identify and evaluate all the key characteristics that impact the performance and energy usage of a stencil computation running on a particular, often heterogenous, computational processing unit. Based on these characteristics we can define a models which minimises the energy usage within a specified computation's deadline of the stencil workload on heterogeneous architectures. Since the problem is computationally intractable, an energy-aware Integer Linear Programming (ILP) formulation can be proposed for finding optimal schedules, but only for relatively simple setups and instances.

In practice, stencil computations are distributed on the large blocks obtained from the decomposition of the computational domain. The computational domain

is a Cartesian grid on which the stencil computations are defined. The optimisation space shows that the best strategy depends not only on load balancing the problem size between the processing units, the processing units specification, and the stencils employed but also on detailed mapping of the communication dependencies of the blocks to the communication topology of respective processing units.

Previous work has yet to attempt to account for the time and energy simultaneously in the context of the distribution of stencil computations between processing units. Thus, the new heuristics that schedule example workloads in real-time are developed, including the communication overhead in the distribution process were proposed. The proposed methods and algorithms have been tested experimentally using multi- and many-core architectures and published in some scientific papers during the research.

1.3 Contributions presented in this thesis

This thesis presents the following key contributions:

- Formulation and presentation of a topology-aware scheduling model for efficient stencil computations executions taking into account different intra-node and inter-node communication links among heterogeneous processors;
- An innovative analytical and methodological approach for modelling and predicting energy usage and runtime for reference stencil patterns on single and multi-node heterogeneous HPC architectures;
- Development and experimental analysis of new heuristic algorithms for simultaneously minimising energy usage and runtime of small and large-scale stencil computations. The proposed energy-aware resource management method considers complex hardware HPC architectures, including different and heterogeneous single and many-node computing setups as well as communication topologies;
- Design of a new Tabu Search algorithm called Task Movement (TM) for efficient solving the given problem;
- Collected results and comprehensive studies of computational experiments comparing TM algorithm performance vs simpler heuristics;
- Development of a flexible and generic scheduling model and heuristics algorithms that can be easily extended or adapted in existing stencil application frameworks adopted in real supercomputing systems;
- Introduction of a set of practical recommendations for application developers and users interested in highly scalable and parallel stencil-based computations.

1.4 Structure of the dissertation

The present thesis is organised as follows:

Chapter 2 introduces the architecture of the state-of-the-art supercomputers based on clusters. We also describe the programming models used to parallelise the applications and tools to measure different metrics connected to performance and energy usage.

Chapter 3 presents the stencil definition and different methods to optimise stencil computations on single and multiple processing units. The performance and energy models are also described.

Chapter 4 recalls basic notions in the computational complexity of combinatorial problems. We provide three examples of linear programs: the integer linear program (ILP), the mixed-integer linear program (MIP) and the binary program (BIP). We also present basic graph theory definitions and depict an edge-colouring problem.

Chapter 5 describes computational experiments that enabled us to discover the critical parameters that impact stencil computations' performance and energy usage. We formulate our problem and define the performance and energy model.

Chapter 6 introduces a method based on ILP to obtain the optimal solution for the formulated problem. We also present the heuristics with two objectives: to minimise the energy usage and load balance of the tasks to meet the deadline. We provide simulation results for the ILP model on a small problem instance.

Chapter 7 presents our implementation of a new Tabu Search inspired algorithm called Task Movement (TM). The simulation experiments include two real-world simulation grids to demonstrate that it is possible to reduce energy usage and improve the overall performance of stencil computations in multi-node HPC setups with different network and communication topologies.

Chapter 8 summarises the research and presents conclusions.

Appendix A provides a dedicated repository to share our algorithms and problem instances with guidelines for running and comparing computational experiments.

Chapter 2
High Performance Computing

This chapter describes the main motivations behind developing energy-saving hardware and software methods to meet the required power budget in HPC. It presents the architecture of the state-of-the-art supercomputers based on clusters, including the processing units, communication between them and network topology. The last sections describe the programming models used to parallelise applications on supercomputers as well as present tools used to measure different metrics connected to the performance and energy usage.

2.1 Architecture overview

High Performance Computing (HPC) has a number of definitions. It may refer to running an application on a dedicated HPC machine or a computing cluster. Generally, HPC aims to get the application running as fast as possible. Primarily, it is useful when the problem is memory demanding and does not fit the memory of a single computer but also when the problem is complex and requires considerable computational power. HPC is utilised in many different areas, such as earth and life sciences, bioinformatics, manufacturing, oil and gas, aerospace and defence, financial services, cyber security or education [84]. In other words, HPC is used when the problem has to be computed in less time, e.g., a model of a new car is tested and designed virtually with more flexible and agile techniques than building an expensive physical prototype. Another example is to complete a specific task before a deadline, e.g., to predict tomorrow's weather today. Finally, HPC may be used to perform a high number of operations where the task has to be scaled, e.g., a service provider requires more computational power to handle changing workloads generated dynamically by end-users.

In the history of HPC, different types of computing systems were designed, e.g., vector computers as an answer to the emergence of computational science, Symmetric Multiprocessing systems (SMP) where the processors distributed between different machines are connected to a single shared memory or cluster systems where a group of tightly connected machines are controlled by a single scheduling software.

The Top500 list contains the fastest systems in the world called supercomputers. According to this list, more than 98% of the supercomputing systems are clusters [110]. The state-of-the-art benchmark used to test the performance of the supercomputers is called Linpack [31]. Linpack is a software package that solves a dense linear system using double-precision calculations. However, this benchmark does not reflect the real-world applications with different workload types. Other benchmarks are under development to address this issue. One example is a data-intensive benchmark called Graph500 [72] that includes three so-called kernels. The first kernel constructs a weighted, undirected graph that subsequent kernels cannot modify. The second kernel performs a breadth-first search of the graph whereas the third one performs single-source shortest path computations. Another example is so-called proxy-applications proposed here [101]. The main goal of these applications is to represent different types of workloads that reflect real-world usage scenarios. One of the proxy applications is Compressible Navier Stokes (CNS) equations with constant viscosity and thermal conductivity that are discretised as stencil computations defined on a Cartesian grid. The Green500 list includes the most power-efficient supercomputers [106]. The top spot on a release from June 2021 of this list holds MN-3 at Preferred Networks, which delivers 29.7 GFLOP per Watt (GFLOP/W) with a performance equal to 1.8 PFLOP/s. Today, one of the main goals of many HPC communities is to develop a supercomputer that will reach one Exa FLoating-point OPeration per second (EFLOP/s) of the performance by the year 2020 within 20MW of the power budget [55], but some of the authors think that the year 2024 is more realistic [56]. The primary obstacle to achieving this goal is power consumption. In order to reach the exascale performance, the system should be able to sustain 50 GFLOP/W, thus the power consumption of the MN-3 supercomputer should be reduced by a factor $2x$. According to the Green 500 list, 13 of the top 15 supercomputers are heterogeneous systems that utilise CPUs and GPUs. Authors in [56] suggest that only these types of systems are able to reach the exascale performance.

There are many different techniques in a hardware design process to improve power consumption, including a well-known process called technology scaling [51]. In this technique the size of the transistors is reduced, and thus the energy usage is decreased linearly. However, the energy cost of moving data still remains constant independently of the transistor size, see Table 2.1.

Table 2.1: Energy consumption of processing unit components

Year	Process size [nm]	Frequency [GHz]	DFMA [pJ]	64b read from 8Kb SRAM [pJ]	Wire energy [fj/bit/mm]
2010	40	1.6	50	14	240
2017	10	2.5	8.7	2.4	150

Other hardware design efforts to improve the energy efficiency for HPC include:

- packaging multiple chip modules on the same silicon;
- 3D stacking of memory and chip;
- power management strategies to dynamically allocate a power supply to the portions of the hardware;
- malleable memory systems to enable the construction of a hierarchy of scratch pads that reside simultaneously with the hardware caches;
- memory models with a relaxed consistency model to facilitate greater memory parallelism and reduce the data movement;
- non-volatile memory.

All the efforts in the hardware are tightly connected and could naturally impact the energy efficiency improvements in the software layer.

The following sections describe different hardware components utilised in HPC heterogeneous cluster systems.

2.1.1 Processing Units

The architecture of the GPU is significantly different from the CPU, see Figure 2.1.

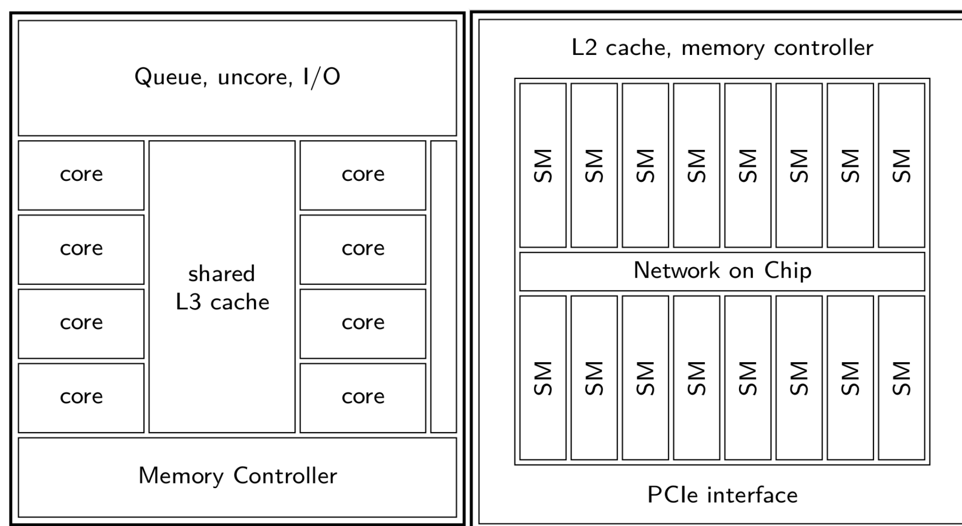


Figure 2.1: Architecture of the CPU (left) and the GPU (right)

GPUs have been developed mainly to render frames that are displayed on a computer screen. However, developing programmable processors called shaders facilitated the implementation of the first algorithms not connected to the graphics rendering. Next, different shaders were merged into unified shaders and enabled General-Purpose Computing on Graphics Processing Units (GPGPU). This unification led to the development of a Compute Unified Device Architecture (CUDA)

programming model [74]. CUDA is an extension of the C++ language that allows for access to the GPU resources. Whereas CPUs have been developed to execute different workload types from the very beginning, GPUs have been focused on rendering. This has implications for performance and energy efficiency. Typically, the GPU can execute more operations per second than the CPU, and each GPU operation requires less energy than the CPU operation. For example, Nvidia Tesla K20m GPU has a peak performance 7x higher than Intel Xeon E5-2670 CPU and 5.76x more memory bandwidth. Nvidia GPU consumes 6x less energy per operation than Intel CPU [26]. However, it is harder to program GPUs as they require a fine-grained parallelisation of the algorithm. Tesla K20m has 2496 cores distributed between 13 Streaming Multiprocessors (SM), whereas Xeon E5-2670 has eight cores. Tesla K20m cores are combined in vectors of 32 called warps and each SM has six warps. Nvidia GPU has L1 and L2 caches that are much smaller than CPU caches. Moreover, Nvidia GPU has a scratchpad cache called a shared memory that must be manually programmed. The slowest memory on the GPU is GDDR5. The computation latency is hidden by multi-threading. On the other hand, the CPU is focused on the performance of a single core and has larger caches that reduce the memory access latency. It has complex control units that improve the throughput of instructions, for example, a branch prediction unit that reduces stalls in the pipeline. Intel Xeon E5-2670 has three large caches (L1, L2 and L3) that reduce the memory access latency. The CPU also has vector instructions called Single Instruction Multiple Data (SIMD). Intel Xeon E5-2670 supports three different vector extensions such as MMX, SSE and AVX. The AVX vector extension has a width of 256 bits and can execute the eight 32 bit values at a time.

The newest GPU architectures significantly increase the theoretical computational power and energy efficiency of double-precision operations. Tesla K20m based on the Kepler architecture provides 1.175 TFLOP/s, whereas a newer Pascal GPU Tesla P100 (PCI version) achieves 4.76 TFLOP/s. The power efficiency increased more than three times from 5.2 GFLOP/W to 18.68 GLFOP/W. This significant increase resulted from a two-generation upgrade where Nvidia skipped Maxwell Tesla cards. The newer Volta architecture with Tesla V100 GPU (PCI version) reaches 7.014 TFLOP/s of the performance and executes 28 GFLOP/W, which represents approximately 50% increase in power efficiency compared to Tesla P100. Volta GPU was introduced only one year after Tesla P100.

2.1.2 Intra-node communication

As described in the previous section, the data on the CPU and GPU have to be moved through a complex memory hierarchy. Several processing units (PUs) can be installed in a single compute node to distribute the computations and execute them efficiently. A configuration where multiple processing units are connected to shared memory is called a shared memory system. The PU transmits the data by writing to

any address in the memory, and other PUs can read from it. Currently, three main types of memory buses may be distinguished for the intra-node communication. For the first CPU-CPU communication type, Intel CPUs utilise Quick Path Interconnect (QPI). QPI in version *1.1* enables connecting up to four CPUs. Each Intel CPU socket may be connected to two QPI links. The QPI link runs at 8.0 Giga Transfer per second (GT/s) in a single direction where each transfer moves two bytes of data [86]. Thus a single QPI link provides 32 GB/s of bandwidth.

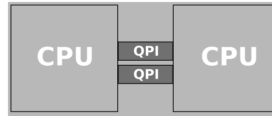


Figure 2.2: Two CPUs connected with two QPI links

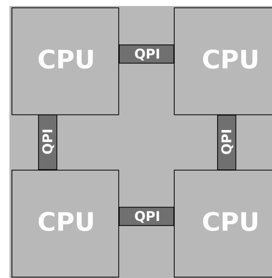


Figure 2.3: Four CPUs connected with four QPI links

Figures 2.2 and 2.3 show the 2xCPU and 4xCPU node configurations. The 2xCPU configuration with two QPI links enables 64GB/s of inter socket communication. However, for the 4xCPU configuration there is no direct connection between CPUs on a diagonal. Thus, CPUs on a diagonal must first move data through the neighbouring CPU and then to the target CPU.

The second intra-node GPU-GPU communication type may be conducted in two different ways. The Nvidia GPUDirect Peer-to-Peer (P2P) technology enables direct communication between GPUs connected to the same Peripheral Component Interconnect Express (PCI Express) Root Complex (RC). PCI Express is a memory bus that connects the CPU with other external components such as GPUs. Typically, Intel CPU has a single PCI Express Root Complex. RC defines a separate hierarchy domain for PCI Express. This hierarchy may be composed of switch components and PCI Express endpoints, see Figure 2.4. Switch aggregates PCI Express endpoints and allows more devices to be attached to a single RC. Figure 2.5 shows connections between four GPUs. Each GPU is attached to the exact RC and communicates using P2P. P2P enables direct access to the GDDR memory of another GPU without going through the CPU's main memory RAM. Figure 2.6 presents the communication between two GPUs connected to different CPUs. Each CPU has a separate RC, and the communication goes through the CPU's main memory. This communication has significantly lower bandwidth than the P2P connection [69].

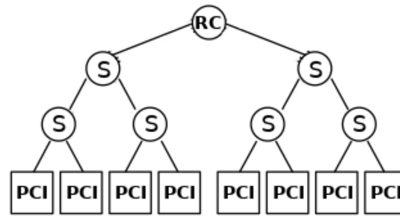


Figure 2.4: Root complex with a PIC Express topology

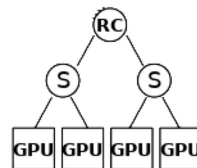


Figure 2.5: Root complex with four GPUs

The third intra-node communication type is between the CPU and the GPU. This communication is done through the PCI Express bus for the GPU based on the Kepler architecture. The PCI Express in version 3.0 with 16 lanes enables moving the data with a bandwidth of 16 GB/s.

For newer architectures such as Pascal and Volta Nvidia introduced NVLink [34]. NVLink enables point-to-point connection between the CPU and the GPU as well as solely between GPUs. Tesla P100 GPU is released in two different variants. The first one, similarly to the Kepler architecture, uses a PCI Express 3.0 bus to connect the GPU to the CPU, while the second one utilises the NVLink bus to enable the connection between the CPU and the GPU. The total bandwidth of NVLink in version 1.0 with four lanes implemented in Tesla P100 is equal to 80 GB/s whereas Tesla V100 utilises NVLink in version 2.0 with six lanes, which provides a total bandwidth equal to 150 GB/s. The NVLink bus also enables connecting of up to four GPUs, see Figure 2.7.

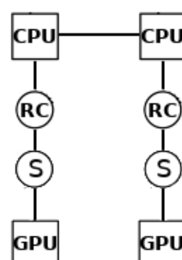


Figure 2.6: Two GPUs connected to two separate CPUs

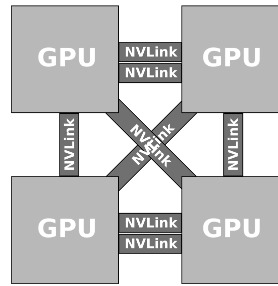


Figure 2.7: NVLink connetcing four GPUs

2.1.3 Inter-node communication

This section describes the communication between computing nodes in an HPC cluster over a network where each PU can transmit data to every other PU by sending and receiving messages. Such configurations are called distributed memory systems. Today, many supercomputers have a hybrid configuration where several PUs are installed in each node and act as a shared memory system, whereas a network is used to connect these nodes. Contemporary HPC clusters utilise different network topologies to connect a large number of nodes such as trees, toroidal meshes and dragonflies.

The advent of large and low-diameter network topologies installed in the most powerful supercomputers impacts the overall performance of many large and highly parallel computations. In the upcoming powerful HPC installations, the critical performance criterion related to minimising costs due to data transmission will play an increasingly vital role. Typically, inter-node communication using a specific network topology for the interconnection of many nodes is usually much slower than intra-node communication. However, parallel applications use conventional read and write operations on memory that processors share within the same node. Therefore, this section describes all the relevant parameters extracted from real HPC network topologies that can be used during the performance optimisation of stencil parallel computations.

We have focused our research on three major network topologies, namely fat-tree, dragonfly and torus, see Table 2.2. They have been selected as they are already adopted in modern multi-node HPC systems listed in [110]. To better understand inter-node and intra-node communication routines, let us present the main characteristics of fat-tree, dragonfly and torus network topologies.

2.1.4 Fat-tree network topology

The fat-tree topology is an example of a tree network in which the computing nodes are connected to a bottom layer of the tree, see Figure 2.8. Each switch has the same number of links going down to its children and going up to its parent. The

Table 2.2: Top500 list from November 2021 of most powerful multi-node HPC systems ranked according to the High-Performance Linpack (HPL) and High Performance Conjugate Gradients (HPCG) benchmarks.

HPL	HPCG	Rmax [PFlop/s]	System/ Location	Interconnect architecture	Processor	Cores	Topology	Power [MW]
1	1	442	Fugaku Japan	Tofu interconnect D	A64FX 48C	7 630 848	Torus	29.9
2	2	148	Summit USA	Dual-rail Mellanox EDR Infiniband	IBM Power9+ Nvidia GV100	2 414 592	Fat tree	10.1
3	4	94	Sierra USA	Dual-rail Mellanox EDR Infiniband	IBM Power9+ Nvidia GV100	1 572 480	Fat tree	7.4
4	16	93	Sunway China	NRCPC	Sunway SW26010	10 649 600	Fat tree	15.3
5	3	70.9	Perlmutter USA	Slingshot-10	AMD EPYC 7763 Nvidia A100	761 856	Fat tree	2.6
6	5	63.4	Selene USA	HDR Infiniband	AMD EPYC 7742 Nvidia A100	555 520	Fat tree	2.6
20	15	21	Piz Daint Switzerland	Cray Aries interconnect	Intel Xeon E5 Nvidia Tesla P100	387 872	Dragonfly	2.4

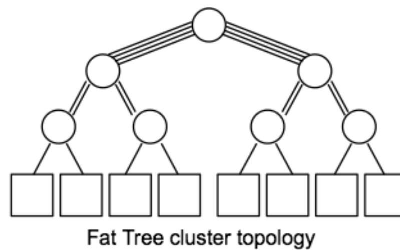


Figure 2.8: Fat-tree network topology

tree gets "fatter" towards the root of the tree, and the switch in the root has the most links compared to other switches.

Typically, it consists of two or three switch levels [79, 62, 48, 113]. In Summit, one of the most powerful HPC systems today, the fat-tree network topology is composed of Dual-rail Mellanox IB EDR 100G with a node injection bandwidth of 23 GB/s. The interconnect is a three-level tree implemented by a switch to connect nodes within each cabinet (first level) and other switches (second and third levels) connecting cabinets. Each IBM POWER9 CPU is directly connected to a network interface card (NIC) using the PCIe Gen4 x8 shared slot. The bandwidth between the CPU and NIC is around 16GB/s, whereas each port on NIC provides 12.5GB/s of bandwidth. The multi-node Summit system contains 4608 nodes. Each node has two IBM POWER9 CPUs and six Nvidia Volta V100 GPUs. 512GB of DDR4 RAM is available for CPUs and 96GB of HBM2 memory for GPUs. Each

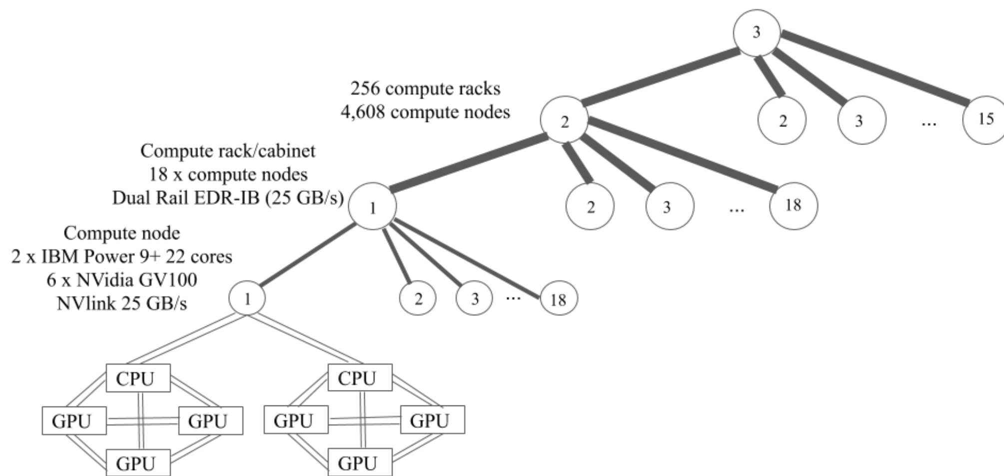


Figure 2.9: Fat-tree network topology implemented in the IBM Power 9+ Summit multi-node HPC system.

CPU with three GPUs composes a group that is connected with NVLink2. The bandwidth between GPUs or between the GPU and the CPU is about 50GB/s in a single direction, see Figure 2.9. The bandwidth between CPUs is 64GB/s. The aggregated bandwidth of eight DDR4 RAM modules is 170.64GB/s, and the aggregated bandwidth of HBM2 is 900GB/s. The POWER9 processor has been built around IBM's SIMD Multi-Core (SMC). The processor provides 22 SMCs clocked at 3.07GHz with separate 32kB L1 data and instruction caches. Pairs of SMCs share a 512kB L2 cache and a 10MB L3 cache. SMCs support Simultaneous Multi-Threading (SMT) up to a level of four, meaning that each physical core supports up to four hardware threads. These threads share the physical core's L1 instruction and data caches. The peak performance of a single processor is up to 540.5 GFLOP/s in double precision. The latest GPU Volta architecture with Tesla V100 reaches 7.8 TFLOP/s of performance in double precision and executes 28 GFLOP/W, which represents approximately 50% increase in power efficiency compared to Tesla P100. Each V100 contains 80 streaming multiprocessors (SMs), 16 GB of high-bandwidth memory (HBM2), and a 6 MB L2 cache available to SMs. The GigaThread Engine distributes work among SMs, and eight 512-bit memory controllers control 16 GB of HBM2 memory.

2.1.5 Torus network topology

The toroidal mesh, also called a torus interconnect, is another example of a network topology used in supercomputers. Each node in a cluster is connected to the adjacent ones in this topology. The signal is routed directly from one node to the other with no need for switches. There are torus networks with different dimensions. In a 1D torus,

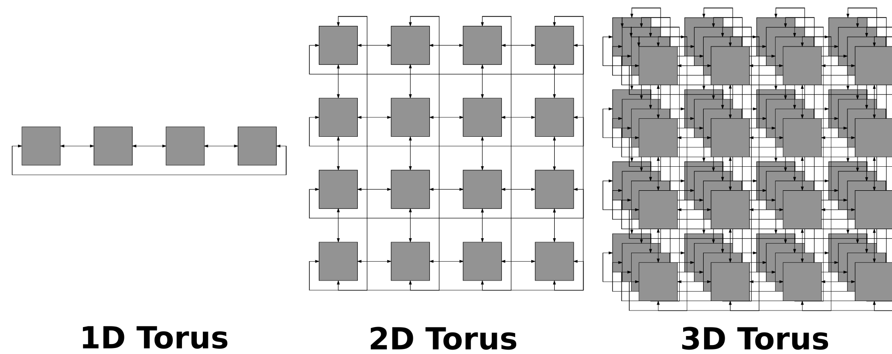


Figure 2.10: Torus network topology with different dimensions

the nodes are connected to two nearest neighbours, and the communication may be applied to two directions. In a 2D torus, the nodes are placed on a rectangular mesh with each node connected to four nearest neighbours. The nodes on the mesh edges are connected to the nodes on the opposite edges. In a 3D torus, the communication takes place in six different directions. Thus, the node can be connected to six other nodes, see Figure 2.10.

The torus topology is a k-ary n-cube network with $N = k * n$ nodes arranged in an n-dimensional grid having k nodes in each dimension. Each node of the torus network is connected to $2 * n$ other nodes. A torus network node can be identified with a unique n-digit radix k address. Network designers determine the properties of the torus network primarily by torus dimensionality (i.e., the number of dimensions, and the number of nodes in each dimension) and link bandwidth. With a high number of torus dimensions per node and limited link bandwidth, the serialisation delay of the packet becomes a significant overhead. Likewise, the network diameter increases with a limited number of torus dimensions and a higher link bandwidth, increasing the end-to-end packet latency. Therefore, when designing a torus network, one must find the right balance of dimensions and channel bandwidth to achieve high performance for the target workloads. Since each torus node is connected to its neighbours via dedicated links, torus networks typically have high throughput for traffic patterns involving nearest-neighbour communication. The torus network concept has been used extensively in the Blue Gene machine and was successfully implemented by Fujitsu in the K-computer machine (6D torus). The torus topology network is implemented in the Post K-computer called Fugaku. Any Fugaku compute node may contain four Core Memory Groups (CMG). Each CMG has 12 computing cores and one assistant core based on the Armv8.2-A series processor, which is clocked at 3GHz connected to 8GB of HBM2 memory. Each core has 512-bit wide SIMD (Single Instruction, Multiple Data) and can execute two FMA instructions (Fused Multiply–Add). The performance of the compute node is over 2.7 TFLOP/s in double precision according to [3]. The aggregated bandwidth of four HBM2 memory modules is 1024GB/s. The torus network topology called TofuD interconnect contains six coordinate axes: X, Y, Z, A, B, and C. There are two possible config-

urations in a rack. The rack contains eight shelves, and each shelf has 48 CPUs. The configuration of the processors in the shelf is 1x1x4x2x3x2 (XxYxZxAxBxC). The top or bottom half of the rack (four shelves) has the following configuration: 2x2x4x2x3x2. The compute node has six network interfaces where the bandwidth of each one is 6.8GB/s; see Figure 2.11.

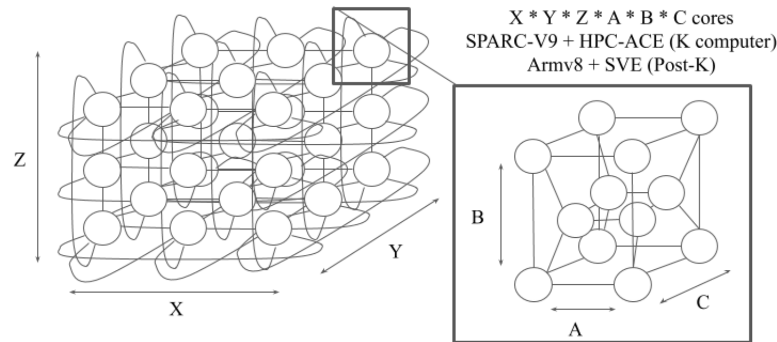


Figure 2.11: Torus network topology implemented in K-computer and Post-K supercomputer.

2.1.6 Dragonfly network topology

A dragonfly topology is a hierarchical network with three levels: routers, groups, and systems [53]. The routers inside each group can have any topology, i.e., fat-tree or 3D torus. The recommendation in [53] is a flattened butterfly [52]. Several groups are connected using all-to-all links, i.e., each group has at least one link directly to each other, see Figure 2.12. This topology focuses on reducing the number of long links and network diameter.

The concept of the dragonfly network topology, a two-level directly connected network, is a good candidate for exascale architectures because of its low diameter and reduced latency, as presented in [71]. Over the last few years, it has been improved in the form of the Slim Fly network topology [112]. The dragonfly topology has been successfully implemented in the Piz-Daint supercomputer and is used to link all compute nodes. Currently, the 5320 multi-node HPC system is connected by the Aries interconnect [6]. Each Piz-Daint compute node contains a single 2.6GHz, 12-core Intel E5-2690v3 Haswell series processor with 64GB of DDR4 RAM [5] and Nvidia Tesla P100 GPU with 16GB of HBM2 RAM where GPU is connected by a PCIe 3.0 x16 bus. The peak performance of a single processor is 499 GFLOP/s in double precision. Tesla P100 GPU reaches 4.67 TFLOP/s of performance in double

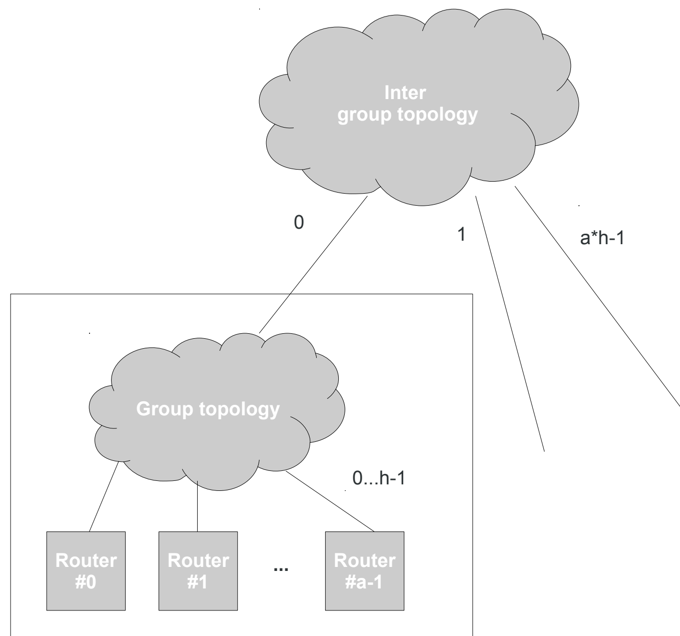


Figure 2.12: Dragonfly network topology

precision. Four compute nodes are connected to each Aries router in this topology and form a compute blade. A chassis has 16 compute blades where three (up to four chassis) form a cabinet, and two cabinets make up a group. The interconnect consists of all-to-all connections among all compute blades in a chassis called the rank 1 network. Compute nodes are connected to the corresponding nodes in different chassis within the group called the rank 2 network. They form 2D all-to-all connections among all nodes in a group. All-to-all connections link groups called the rank 3 network. Minimal routes between any two nodes in a group are two hops. The non-minimal route requires up to four hops, as presented in Figure 2.13.

The performance and energy usage of network topologies mainly depend on the number of intermediate hops that a message has to go through in order to reach the target node. These hops influence the latency and bandwidth of the connection links. Hops can represent a switch in a fat-tree topology or a network adapter in a torus topology. Another parameter that influences the efficiency of topologies is congestion which consists of heavy traffic that blocks any number of internal paths within an interconnection network. Typically, the congestion results from contention where several packets from different input ports concurrently request access to the same output port. Only a single packet can be sent while the other packets wait in a queue until the output port becomes available. When the contention persists, the queues are filled up and block the packets coming from the source switches. This congestion may eventually be spread across various paths. As a result, the average latency may be increased, and the network throughput may be decreased. Many different techniques are proposed to address increased latency, for example, congestion management in a fat-tree topology based on an Infiniband interconnection

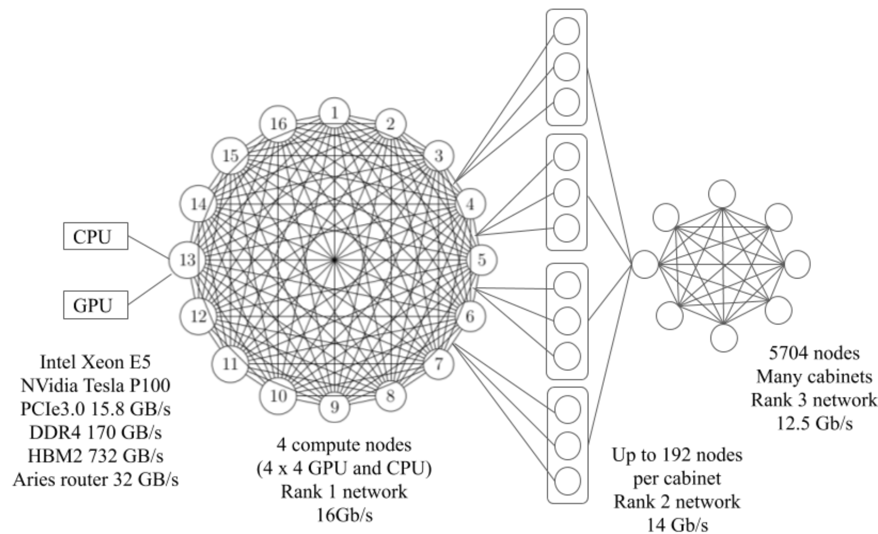


Figure 2.13: Dragonfly network topology implemented in Cray XC50 Piz-Daint multi-node HPC system.

technology [32].

2.2 Parallel application programming and execution environments for stencil computations

The current generations of most powerful HPC systems are already hierarchical in interconnection, communication, memory architecture, cache level, coherency, and non-uniform access. Future generations of exascale HPC machines will feature even more complex hierarchies. However, over the last decades, general-purpose HPC systems have supported various parallel programming environments. Application developers have been provided with parallel programming environments as an abstraction layer necessary to obtain the required level of concurrency and parallelism. In general, it can be achieved by decomposing stencil computations by expressing all the critical requirements related to data locality within the source code. The decomposition process itself can be done in three different ways:

- application profiling, using appropriate software analysis and optimisation tools to detect some regularities in the stencil software source code (data-parallel model)
- identifying procedures that meet specific functionalities in the stencil source code. Then, identified functionalities form a set of smaller entities (tasks) that can be scheduled and executed in parallel on multi-node computing resources

(task-parallel model)

- automatic parallelisation, scheduling and tuning during the stencil code compilation process without the developer's intervention in the code.

Stencil code developers used to delegate scheduling and management functions to the underlying software layer, which automatically guaranteed the acceptable performance efficiency and portability of the code. However, many manual changes supported by various optimisation techniques before and during the execution are still required within the stencil source code. The main reason is the increased memory and communication hierarchies in HPC systems, as discussed in the previous section. System architectures at exascale are unlikely to guarantee a massively parallel application of complete coherence in cache access. Naturally, it will directly impact model changes in parallel programming environments, taking into account the physical location of data in the microprocessor architecture.

2.2.1 Multi-node parallel programming environments

To hide the underlying complexity of heterogeneous multi-node HPC systems for efficient stencil code execution in a parallel mode is a challenging task. The decomposition approach based on automatic parallelisation is limited in practice. Many automatic tuning options have been optimised for a specific processor microarchitecture. Thus, the software performance portability from one specific hardware architecture to another is also naturally restricted. Nevertheless, aiming at pre-exascale and exascale multi-node HPC systems, application developers must carefully select application decomposition techniques. Additionally, they should consider various scheduling attributes related not only to heterogeneous computing parts, see for instance [78], but also to the hierarchical memory allocation, data placement, data movement, and communication. There are some promising research activities in this area, for example, recent successful experiments using a compiler-based autotuning framework for the geometric multigrid linear solvers presented in [9]. From the perspective of parallel stencil code synchronisation in many existing multi-node HPC systems, we can still distinguish two major programming models and corresponding execution environments:

- The Message Passing model provides developers with a distributed memory structure assuming that only the local memory cache is directly accessible to the task (process). Communication with other tasks (processes) is carried out by data exchange in messages. The primary and commonly accepted standard in high-performance computing is the Message Passing Interface (MPI), the standard for message exchange among multiple tasks (processes) developed back in the 1990s [41]. It is now most commonly accepted and supported by hardware vendors on multi-node HPC installations together with various valuable extensions [44], e.g., support for distributed computing systems [1], The

MPI standard in version 3.0 includes neighbourhood collectives as sparse communication patterns such as 3D Cartesian neighbourhoods that occur in stencil computations to improve performance and portability of applications [45]. It may replace the traditional point-to-point communications typically utilised in many implementations of stencil computations. It is still an open question if the message passing is the correct paradigm for all systems. However, some authors suggest that this may not be true, especially for hierarchical HPC systems [83, 70, 61]. Basic MPI ignores the fact that today's computational nodes contain multi- and many-core processors that reside on a shared memory within a node. Furthermore, as the energy cost of moving data across the interconnect is substantially higher than the cost of intra-chip communication, one can expect that more emphasis will be put on shared memory programming models [91]. Although some MPI libraries and execution environments employ shared caches within a node to improve communication time, these optimisations are usually hidden from the application programmer.

- The Shared Memory model provides an entire address space in the memory. Thus, developers at the level of APIs can quickly scale the data size necessary to perform calculations. A widely accepted standard is Open Multi-Processing (OpenMP), a standard for programming applications that allows creating computer simulations for computing nodes with shared memory [20]. The OpenMP standard enables developing performance portable applications that would run on multi-core CPUs. The application that employs it is easy to maintain and debug as it can still run as a valid serial code. Another relevant parallel programming paradigm is called Partitioned Global Address Space (PGAS) [30]. Unlike the OpenMP programming environment, in the PGAS paradigm all variables in a specific local memory area are private for a given thread. Consequently, developers can mark the memory space as shared for other threads to read or modify. They can consider the hierarchical memory structure on a heterogeneous computing node. Then, they can optimise the application performance accordingly by taking into account data locality as it was demonstrated, for instance, in [18].

Additionally, we have observed the emergence of new hybrid programming and execution environments in multi-node HPC systems. They can use the increasing multi-node computing power by effective combination of MPI mechanisms (for programming data exchange between computing nodes in a multi-node computing setup) with OpenMP directives (for programming shared memory). Additionally, many existing heterogeneous computing nodes in HPC systems locally support one of the following environments:

- CUDA - a high-level programming environment based on the C programming language that is an integral part of the universal architecture of multi-core processors (most commonly used for accelerators and graphics cards)

- Open Computing Language (OpenCL) - a high-level programming environment based on the C programming language supporting the development of applications operating on heterogeneous computing nodes consisting of CPU and GPU units
- OpenACC, SYCL and C ++ AMP - programming environments are extending the CUDA and OpenCL environments with appropriate improvements in programming interfaces. It helps developers analyse and optimise the parallel code on various accelerated computing nodes

Many exascale supercomputers will probably be capable of running and supporting various software stacks based on the hybrid *MPI+X* model [10]. In this case, X denotes one of the above-mentioned parallel programming environments optimised for the specific hardware architecture within a specific computing node [2]. Therefore, the hybrid MPI combinations with CUDA, OpenCL, and OpenACC are promising for stencil computations. Considering specific configurations of powerful HPC systems and typical operations in stencil-based computer simulations, some software frameworks are worth considering. For example, an exciting solution is the hybrid programming and runtime environment called StarPU, which supports job scheduling on multi-node GPU-based HPC systems [8]. Another solution named StarSS provides a unified execution model for heterogeneous jobs supporting different scheduling algorithms. It supports essential load balancing and data management mechanisms for processors connected to multiple GPUs [80]. Some useful extensions to StarSs were proposed as OmpSs in the form of OpenMP-like pragmas, and it can also incorporate the use of OpenCL or CUDA kernels. OmpSs supports different scheduling strategies and offers an advanced runtime system to schedule tasks efficiently as demonstrated in [81].

The CUDA programming model has been selected in this thesis as a more mature standard for Nvidia GPUs. CUDA is an extension of the C++ language that enables access to GPU resources. A function executed on the GPU is called a kernel. A thread computes the work described in the kernel. The CUDA programming model defines a particular thread hierarchy where threads are grouped into blocks, and each block is executed on SM. A group of blocks forms a one-, two- or three-dimensional grid.

In the basic MPI model adding new cores significantly increases the communication load, relatively to the local work size on a per rank basis. The hybrid MPI+OpenMP approach has been utilised in this thesis to address the communication load and to implement and run stencil computations on CPUs efficiently. By finally adding the GPU, the hybrid MPI+OpenMP+CUDA programming has been selected as a good model to benchmark stencil computations described in Section 3.2.2 for an HPC heterogeneous cluster with both CPUs and GPUs taking advantage of the distributed memory inter-node parallelism and the shared memory intra-node parallelism.

2.2.2 Domain Specific Languages for stencil computations

Many research efforts have been dedicated to developing, deploying, and testing multi-node HPC systems characteristic for parallel high-level domain-specific programming environments called Domain-Specific Languages (DSLs). One of the basic assumptions for DSL environments is to hide the complexity of underlying multiple heterogeneous computing nodes for stencil computations as much as possible. Today, there are many DSL frameworks available for stencil computations, e.g., Mint [100], Physis [67] or recently released YASK [116] and PSkel [78]. In practice, many stencil problem sizes become significantly larger than the fast-memory capacity available on a many-core processors node. Thus, the sequential time-step algorithms create an overwhelming number of misses from the fast-memory shared cache, considerably degrading performance. The new multi-level temporal tiling approach for efficient HPC stencil computation is demonstrated in [115].

DSL aims to eliminate many disadvantages offered by software frameworks, particularly those using the most popular hybrid *MPI + X* model described in the previous subsection. In the context of the considered scheduling problem for stencil computations, it is worth mentioning our contribution to the development of a DSL framework called CaKernel. It enabled efficient execution of stencil computations on more complex heterogeneous architectures, as presented in [13]. Further improvements of the CaKernel framework resulted in an extended DSL called Chemora optimised for solving systems of Partial Differential Equations (PDEs), which targets modern HPC architectures [14]. Chemora was based on Cactus that sees prominent usage in the computational relativistic astrophysics community [88].

Additionally, we have developed a high-level stencil framework implemented for the EULerian or LAGrangian model (EULAG)[82]. EULAG is an anelastic model for simulating low Mach number flows under gravity. It was executed efficiently using the hybrid *MPI + X* environment on heterogeneous multi-node HPC systems, as we demonstrated in [24]. We provided various task scheduling methods using the hybrid *MPI + X* environment. The proposed DSL framework for stencils was written with C++ templates and provided a portable code with no need for additional dependencies. The C++ templates with the static domain decomposition allowed a compiler to efficiently optimize the prepared stencil code. The proposed flexible domain decomposition scheme with the subdomain partition to fit the memory hierarchy supported load balancing between an arbitrary number of CPUs and GPUs.

2.3 Selected tools

This section presents tools used to measure different metrics connected to the performance and energy usage of the application.

2.3.1 Performance measurement

To obtain performance metrics for the considered stencil computations running concurrently on CPUs and GPUs, two tools have been used: Intel Advisor [107] and Nvidia Profiler [109], respectively. Intel Advisor allows to quickly generate reports and find hot spots in the application code, for example, loops that occupy a significant part of the execution time. These reports are called Surveys as they add minimal overhead to execution time and require no modification in the application code. Different parameters can be collected for each loop: the number of executed FLOPs, the number of bytes moved between the CPU and the memory hierarchy (including L1, L2, LLC and DRAM), arithmetic intensity I, the number of FLOP per iteration, the number of bytes moved per iteration, the number of iterations, and utilisation of vectorisation. This report is called "Find trip counts and FLOPS" and adds significant overhead to the execution time (typically ten times longer than execution time).

The second tool, Nvidia Profiler, displays a timeline activity of the GPU together with the CPU and provides the analysis of data flow through memory buses. It is instrumental in verifying the time spent on the communication between the CPU and the GPU as well as the data locality on the GPU. These two tools have been essential to collect metrics needed to prepare and verify the performance model of the stencil computations on both CPUs and GPUs.

2.3.2 Energy measurement

The precise measurement of energy consumption of HPC resources and application is a significant challenge in preparing an energy model that can be used for energy-aware scheduling. The need for fine-grained energy measurements caused the manufacturers of PUs to implement different software interfaces to measure the energy consumption. The Running Average Power Limit (RAPL) [43] is an Intel API written in a C language, which allows measuring the energy usage and controlling the power budget of the different components of Intel CPUs such as socket, core, uncore (only for desktop), and DRAM (only for the server solutions). This API supports Intel processors starting from the Sandy Bridge architecture which appeared in 2011. The measurement frequency of this API is equal to 1000Hz. The default granularity of the energy measurement is equal to 15.3uJ. On multi-core CPUs, the frequency switching implicitly changes an operating voltage. The voltage is optimised by the hardware based on several factors. All processor cores that execute

some workload share the same frequency and voltage. The multiple frequency and voltage pairs while executing the code are called *P-states*. The idle cores switch to low-power idle states called *C-states*. At the higher levels of *C-states* power saving actions are taken, such as flushing the caches, stopping the clocks, and reducing the voltage to zero. These hardware optimisations reduce energy usage and are taken into account during the measurements.

Nvidia provides the Nvidia Management Library (NVML) API [108] for its GPUs. This interface supports GPUs starting from the Fermi architecture and can only be utilised in the server solutions. The NVML interface allows users to manage the power states of GPU and measure the power usage of the whole board in mW. Measurement frequency can be controlled. Both APIs are used during the computational experiments described in Section 5.1.

Chapter 3

Stencil computations

This chapter gives a brief introduction to stencil computations. The stencil definition is provided with examples of two stencil computations representing the three-dimensional data access patterns. Different methods to optimise the performance of stencil computations on single and multiple processing units as well as different domain distribution algorithms are characterised. The last sections describe state of the art across performance and energy models.

3.1 Definition

A stencil computation is defined on a multi-dimensional structural grid where each point on a grid is updated with a strict pattern. The pattern defines which neighbouring points are used during a stencil computation. Let D_t be the three-dimensional grid characterized by the (i, j, k) indices for a time step t. The currently updated point is defined as $D_{t+1}(i, j, k)$ whereas the neighbouring point is represented as $D_t(i + \alpha, j + \beta, k + \gamma)$. The $\alpha, \beta, \gamma \in I$ values describe the shift in each direction from the currently updated position where I is a set of integral values. A single update of the whole grid is called a time step. In this study the focus is on an explicit method where a current time step $t + 1$ is updated by using values of the grid points from a previous time step t. Figure 3.1 shows an example of a seven-point

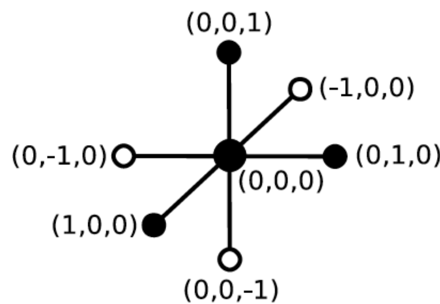


Figure 3.1: Seven-point stencil

stencil where the values of the six neighbouring points are accessed and applied to

the value of the currently updated point. This stencil is represented by the following equation:

$$D_{t+1}(i,j,k) = \theta * (D_t(i,j,k) + D_t(i+1,j,k) + D_t(i-1,j,k) + D_t(i,j+1,k) + D_{t+1}(i,j-1,k) + D_t(i,j,k+1) + D_t(i,j,k-1)) \tag{3.1}$$

A destination grid D_{t+1} for a time step $t+1$ is updated with a stencil that loads the seven values from the source gird D_t for a time step t. The values are fetched from the six directions $i+1$, $i-1$, $j+1$, $j-1$, $k+1$, $k-1$ as well as the middle point and multiplied by a constant θ. Another example is a twenty-seven point stencil shown in Figure 3.2.

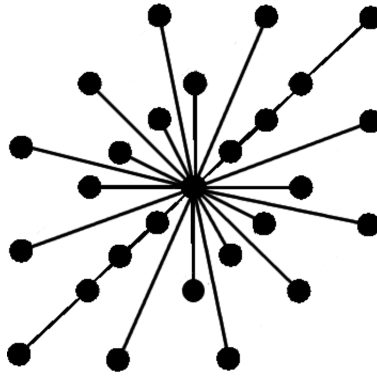


Figure 3.2: Twenty-seven point stencil

This stencil involves neighbouring points from all directions, including the edge, corner and face points; see the equation below:

$$\begin{aligned} D_{t+1}(i,j,k) = \theta * (D_t(i,j,k)+ \\ D_t(i-1,j-1,k-1) + D_t(i-1,j-1,k) + D_t(i-1,j-1,k+1)+ \\ D_t(i,j-1,k-1) + D_t(i,j-1,k) + D_t(i,j-1,k+1)+ \\ D_t(i+1,j-1,k-1) + D_t(i+1,j-1,k) + D_t(i+1,j-1,k+1)+ \\ D_t(i-1,j,k-1) + D_t(i-1,j,k) + D_t(i-1,j,k+1)+ \\ D_t(i,j,k-1) + D_t(i,j,k) + D_t(i,j,k+1)+ \\ D_t(i+1,j,k-1) + D_t(i+1,j,k) + D_t(i+1,j,k+1)+ \\ D_t(i-1,j+1,k-1) + D_t(i-1,j+1,k) + D_t(i-1,j+1,k+1)+ \\ D_t(i,j+1,k-1) + D_t(i,j+1,k) + D_t(i,j+1,k+1)+ \\ D_t(i+1,j+1,k-1) + D_t(i+1,j+1,k) + D_t(i+1,j+1,k+1)) \end{aligned} \tag{3.2}$$

Similarly to the seven-point stencil, all the values are summed up and multiplied by a constant θ. Both stencil types are utilized in this dissertation as they are the important examples of 3D stencils. The seven-point stencil is used as the state-of-the-art benchmark [28] for the performance of stencil computations whereas the

twenty-seven point stencil is a good representative of the data demanding access pattern [29]. Stencil computations may be further divided into two categories depending on the spatial position of the updated point within the grid: the inner computations and the boundary computations, see Figure 3.3.

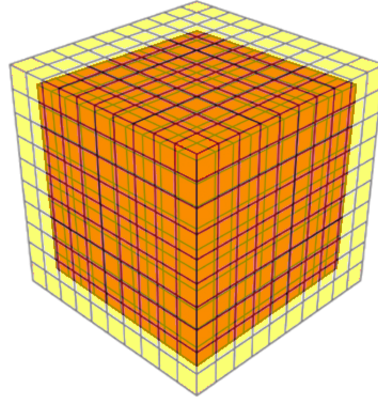


Figure 3.3: The example 3D grid where the red colour shows the interior and the yellow colour shows boundaries

The inner computations have the uniform pattern across the grid and are applied to the interior of the grid, whereas the boundary computations are applied at the boundaries of the grid and may have a different specialised pattern for each boundary depending on the application in which the stencil computation is used, for example, the Dirichlet or Neumann boundary conditions [33].

In this work, the focus is on the stencil computations applied to the inner area of the grid. These stencils are widely utilised in different applications and have a uniform stencil pattern across a grid. Moreover, for a particular grid much larger than the capacity of the available memory of a single processing unit, the computations of stencils defined on an inner area typically require significantly more computing power than the stencils defined on the boundaries.

3.2 Performance optimisation methods

In general, the considered stencil computations perform global sweeps through data structures typically much larger than the capacity of the available data caches within processing units. Additionally, accessing data in the main memory within the hardware is not fast enough, and there is often a bottleneck between the local cache and the main memory. Therefore, many researchers have already tried to exploit data locality in stencil computations by performing operations on cache-sized blocks of data after domain decomposition [90] after time decomposition [35] or proposed cache-aware optimisation algorithms on many-core modern processors [73]. Some of the authors assume that within a single time step there are only stencil compu-

tations [35]. This assumption allows for application of the optimisation technique called temporal parallelisation.

In this study, the focus is on a workflow of the application, e.g., the computational fluid dynamics (CFD) simulations, where different types of computations are mixed within a single time step, such as the implicit methods, reductions, pointwise calculations, and stencil computations. Therefore, it is not possible to utilise temporal parallelisation.

Some frameworks try to ease the implementation of stencil calculations on multiple processing units. A user develops a single stencil code in a framework's specific language which then, during a compilation phase, is translated to a target computing architecture. The frameworks distribute the computations to employ multiple processors. The distribution involves the decomposition of the Cartesian grid into overlapping blocks. The overlap, called the halo region, is needed to update a decomposed block on borders correctly. A single processing unit updates each block. The minimal size of the overlap depends on a stencil pattern. For example, Physis [67] uniformly decomposes a global domain over all the accelerators as instructed by a user-controllable parameter. The user has to determine which decomposition provides the best performance experimentally. The Physis framework focuses only on the GPU architecture. Similarly, a Chemora framework utilises a simple decomposition method by a uniform partition where each CPU and GPU receives blocks of the same size [14]. On the other hand, authors in [78] present a new method that allows programmers to partition the data contiguously between the central processing unit and accelerators within a single computing node. In contrast to our approach, their method does not allow finding an optimal distribution of the domain between heterogeneous computing architectures in terms of time and energy costs. What is more, there is a lack of advanced analyses of stencil optimisations and performance modelling connecting specific properties, such as communication and data locality, together with architectural time and energy costs. In order to balance the grid points between heterogeneous processing units, our approaches take into account different speeds of the processing units. Moreover, the data dependencies between grid points based on the stencil pattern require data exchange between the processing units. The communication of these data through different communication buses may have a significant performance cost. Thus, while assigning grid points to processing units one has to not only balance the computational load but also carefully distribute the data to minimise the communication cost.

The following two sections describe the applied parallelisation methods of stencil computations on single and multiple processing units. The primary purpose of designing and implementing these new methods is two-fold: to provide a good reference to validate the solution quality developed vs optimal one in the model described in Section 6.1 and to determine all the parameters required by time and energy models described in Section 5.3.

3.2.1 Single processing unit

The well-known 2.5D blocking method is employed [73] to implement stencil computations on both CPU and GPU architectures. This optimisation method with a manually tuned procedure ensures good performance and avoids additional overheads imposed by the parallelisation frameworks. It is essential for the acquisition of the selected metrics for the performance and energy models described in Section 5.1. The three-dimensional grid D_t is divided in a (i, j) plane and computations are streamed through the third dimension k. The obtained column represents the data with the size of the layer equal to $dim_i x dim_j$. The algorithm requires storing the $2R + 1$ layers simultaneously, where R is a radius defined as the number of gird points accessed by the stencil in the k dimension. For example, the seven-point stencil mentioned in Section 3.1 has R equal to 1, and thus the number of layers is equal to 3. The stencil computation is based on intelligent placement of the layers in the buffer, representing the processing unit's registers and cache. The size of the buffer is equal to the number of layers. First, the $2R + 1$ layers are loaded from the main memory to the buffer. Then, for each iteration through the k dimension the computation is as follows (see Figure 3.4):

1. perform the stencil computation at the position 1 in the buffer;
2. move the computed value from the position 1 to the position 0 in the buffer;
3. store the value at the position 0 in the main memory;
4. move the value from the position 2 to the position 1 in the buffer;
5. load the value from the main memory to the position 2 in the buffer.

This method allows reducing the memory requirements as only $2R + 1$ layers are needed simultaneously instead of the whole column. It is beneficial for the existing GPUs where the cache size is relatively small compared to CPUs.

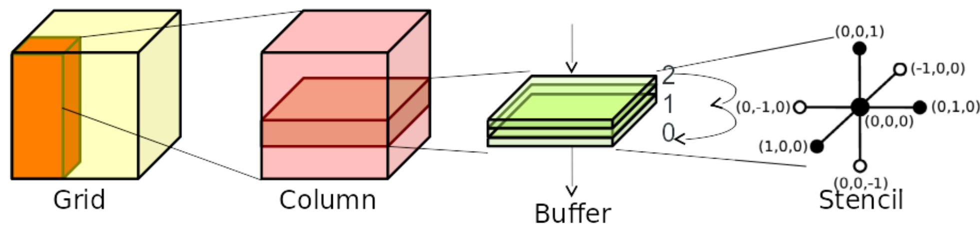


Figure 3.4: The generic view of the 2.5D blocking method

3.2.2 Multiple processing units

There are many different decomposition strategies available. Some use an MPI-all parallelisation scheme with a uniform partition where each core of the CPU maps to a single MPI process with no utilisation of the shared memory on a compute node.

This scheme is straightforward to implement and run as it requires no knowledge about the Non-Uniform Access Memory (NUMA) topology of the underlying hardware [59]. On the other hand, the number of MPI messages required to exchange is a multiple of the cores, leading to substantial communication overhead. Another choice is a strategy that assigns a single MPI process to the whole computing node. It decomposes the obtained block for a specified number of processors (CPUs) and accelerators (GPUs) on the node and minimises the number of MPI processes, hence the communication overhead, as described in [114]. The drawback of this method is that the inner part of the block is only decomposed in one dimension. Therefore, it is not flexible in balancing the load between the accelerators and processors on the node. Another strategy performs a uniform decomposition where both CPU and GPU are mapped to a single MPI process. In this case, the subdomain boundaries are updated and communicated by the CPU, whereas the GPU handles the inner points. In this approach, the CPU serves as a management entity and does not execute any computations; consequently, this strategy uses CPUs inefficiently.

In this thesis, the main idea behind the parallelisation of stencil computations between the processing units is based on data decomposition, where each processing unit updates the fixed part of the grid called a block [24]. As stencil computations generally require neighbour points to update a point, the boundaries of the blocks have to be communicated between processing units. The communicated boundaries are saved in a designated buffer called the halo region, see Figure 3.5.

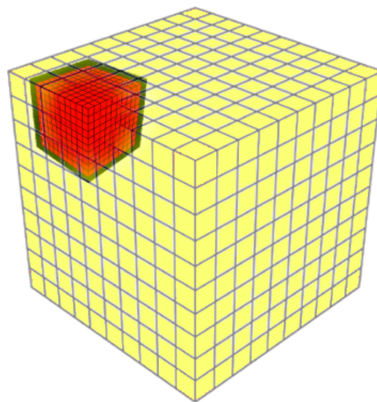


Figure 3.5: Subgrid with the halo region

The scheme can partition the domain into non-uniform blocks for the arbitrary number of processors and accelerators in all three dimensions. This scheme enables computing on different cluster configurations of CPUs and GPUs within the computing node. According to achieved results presented in [24], our approach can efficiently decompose the domain on the CPU-only clusters including the NUMA machines, architectures with global shared memory, on the GPU-only clusters with fat nodes containing one GPU per single CPU core as well as on the hybrid clusters with powerful CPUs and GPUs. In order to efficiently utilise the data locality, the OpenMP and MPI models are employed for the intra-node and inter-node commu-

nication, respectively. Each CPU and GPU has assigned separate MPI processes to the selected cores. The GPU parallelisation is done using the CUDA programming model, whereas the CPU parallelisation employs the OpenMP model.

The partitioning mechanism is performed before the compilation of the code for a target computing architecture. Thus, the obtained decomposition is static during computations. The static decomposition allows the compiler to optimise the code for stencil loops by utilising various techniques, such as loop unrolling and vectorisation. Once the block for each processor is obtained, it is further decomposed to the optimal size for the cache blocking and receiving the optimal size for the given processor. All the details of the block decomposition are described in the previous Section 3.2.1.

The load balancing procedure is critical to find a suitable partition of the computational domain between the heterogeneous computing resources. Another essential assumption is to finish all the computations before a specific deadline. The parallelisation method of stencil computations on multiple processing units to measure selected metrics is further described in Section 5.1.

3.3 Performance models

Performance models for modern heterogeneous processing units using sophisticated memory hierarchies should be as efficient and straightforward as possible to explore the properties of advanced hardware units. One of the commonly used performance models is the Roofline model which allows developers to analyse and predict application performance based on a processing unit computation and memory capabilities [111]. In a nutshell, the application is modelled as a ratio of arithmetic operations to the number of bytes sent through the memory hierarchy. The performance of a basic von Neumann architecture that contains two levels of memory hierarchy can be predicted with the Roofline model. However, the model can be extended to support a more complex memory hierarchy with multi-level caches [98].

It is essential to determine metrics for the Roofline model correctly. The work in [76] proposed to use the Hardware Performance Counters [7] to find a number of floating-point operations executed and a number of bytes moved by the given application. However, the authors showed that such methodology is not precise for the latest generations of processing units. Another interesting work that utilised the Roofline model was presented in [17]. The authors estimated the performance of applications based on the scheduling of a computation directed acyclic graph (DAG) on a model of a microarchitecture and extracted from it the data concerning utilisation of the resources. First, the source code and characteristics that describe the properties of the target architecture are the input parameters. Then, there is an intermediate step of compiling the code to the LLVM intermediate representation (IR) [60]. The nodes of computation DAG are the instructions of the LLVM IR, including both computations and memory instructions. Finally, they execute the

IR of the application in the modified LLVM interpreter to build a schedule of the nodes in the computation DAG. The performance estimation provides the execution procedure. The advantage of this approach is that it considers the capacity of the caches and the size of input data to determine the application performance. The main disadvantage is a lack of accounting for the vectorisation and the fact that each generated roof for each type of the nodes is specific to the given application and its input.

The following two sections describe the Roofline model and its extension in more detail relevant for our research. These models are enhanced to support the stencil computations, see Section 5.3.

3.3.1 Roofline model

The Roofline model assumes a relatively simple paradigm of the two-level memory hierarchy where data come from the slow memory (i.e. DRAM), see Figure 3.6.

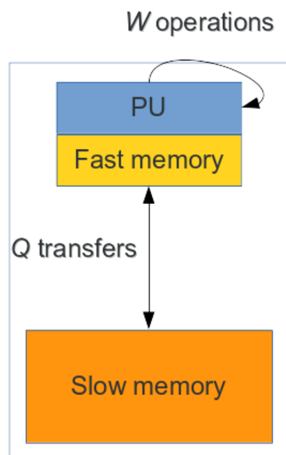


Figure 3.6: Architecture with a two-level memory hierarchy

DRAM is the slowest data path utilised in many existing processing units that limit the amount of data sent in a given time bs. This data path is the first constraint called roof that limits the application performance. The other one is the applicable peak performance P_{max} of the processing unit that fetches data from the fast memory (i.e. the L1 cache). The type of roof that constrains the application performance is selected according to the value of a computational intensity I. The computational intensity is the work W per byte transferred Q_{MEM} over the slowest utilised data path. It is assumed that the work is defined as the number of floating-point operations. The W and Q_{MEM} values are measured for the given application, whereas P_{max} and bs are obtained from the processing unit, see equations below:

$$I = W/Q \tag{3.3}$$

$$P = min(P_{max}, I * bs) \tag{3.4}$$

The P variable represents a modelled performance by selecting which element of the processing unit is constraining the overall performance. In other words, it denotes whether the application is compute-bound or memory-bound, see Figure 3.7.

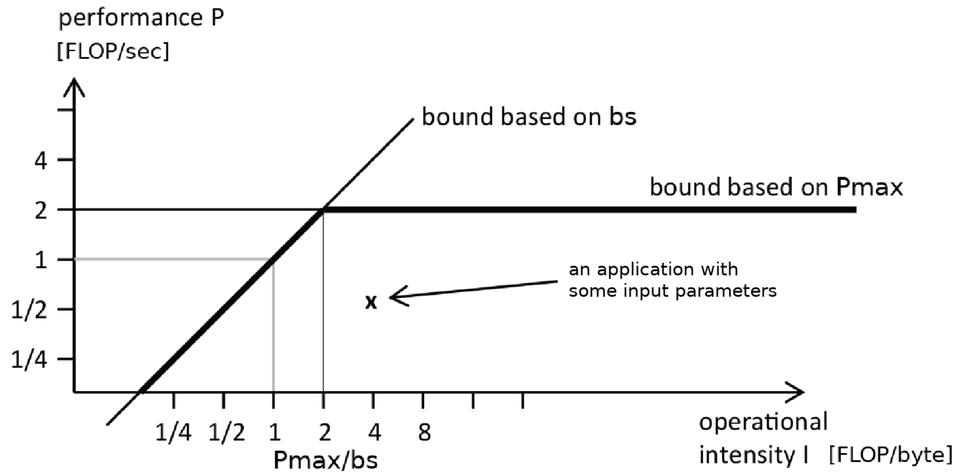


Figure 3.7: The Roofline model

If the application has a computational intensity below 2, it is memory-bound. Otherwise, it is compute-bound. Figure 3.8 shows the placement of the four well know computations on the Roofline graph:

- daxpy is $\alpha * x + y$,
- dgemv is $y = \alpha * A * x + \beta * y$,
- dgemm is $C = \alpha * A * B + \beta * C$,
- FFT is Fast Fourier Transform,

where α and β are the scalars, x and y are vectors, A, B and C are matrices.

Based on this graph, one can see that the FFT and dgemm computations are compute-bound, whereas the daxpy and dgemv computations are memory-bound.

The model compromises the following assumptions:

- the effective bandwidth can be determined with software benchmarks;
- the data transfer and computations perfectly overlap;
- the data path to the main memory is only modelled while all other memories, e.g., caches, are infinitely fast;
- the bandwidth of the slowest data path can be utilised up to 100%.

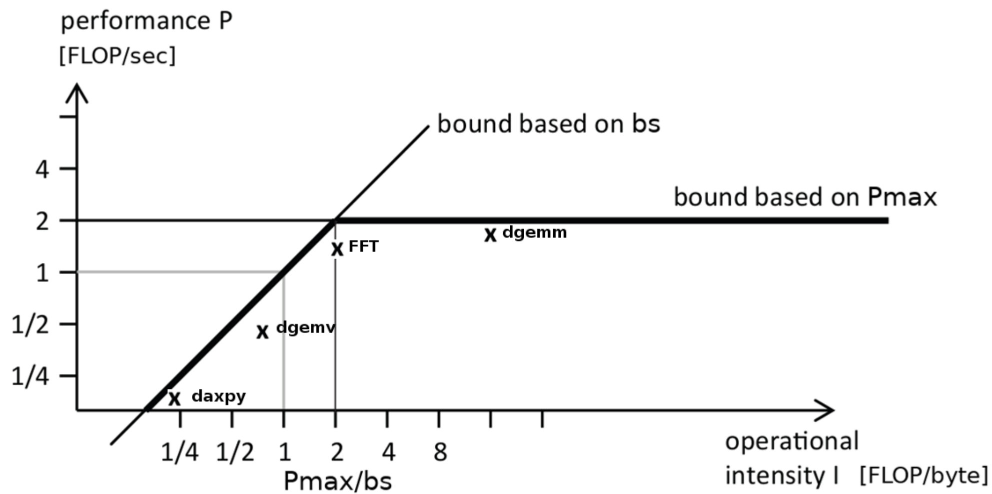


Figure 3.8: Four different computation types on the Roofline model diagram

3.3.2 Cache Aware Roofline model

The Roofline model accounts for only the slowest data path, whereas all other memory levels are assumed to be infinitely fast. The consequence of this approach is that it is hard to predict the computational intensity for a given application, see Figure 3.9.

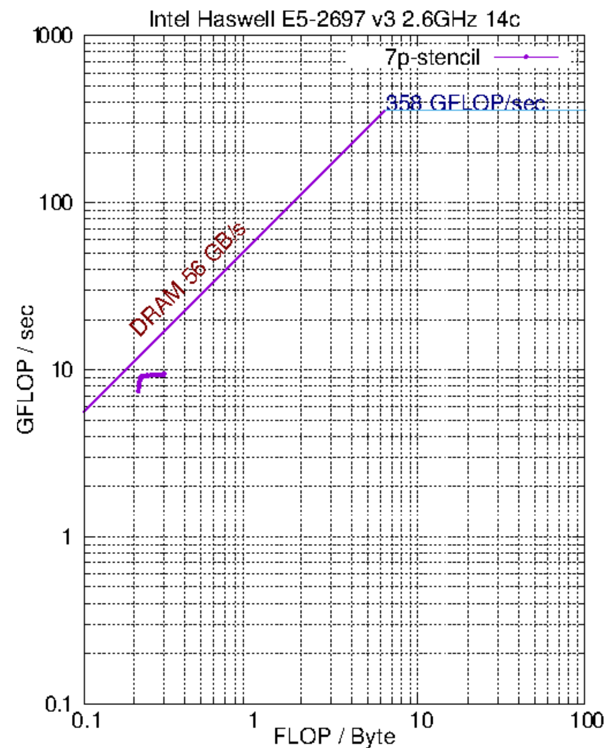


Figure 3.9: Roofline model of an example application

This figure shows the measurements of a seven-point stencil with different input

grid sizes on Intel Xeon E5-2697v3 CPU. One can notice that by increasing the grid size, both the computational intensity and the overall performance decrease. This is because the number of bytes moved from the slowest memory (DRAM) increases faster than the number of executed floating-point operations. One can expect that all the data used are cached for a small input size with a low number of grid points. With the increasing number of grid points, the data sets can not fit within the caches and must be fetched from DRAM. In order to verify expectations, distribution of data in the high-level caches must be carefully explored.

The Roofline model is extended to include the number of bytes Q moved through the memory to higher cache levels [47], see Figure 3.10.

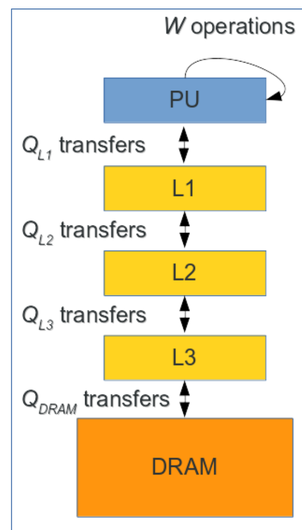


Figure 3.10: Cache aware Roofline model

For example, the calculation of the bytes transferred for the CPU with three levels of caches is the following:

$$Q = Q_{L1} + Q_{L2} + Q_{L3} + Q_{DRAM} \tag{3.5}$$

Figure 3.11 presents measurements of the Spherical Harmonics dwarf using the same input as in Figure 3.9.

The computational intensity is the same regardless of the number of fields used or the test case type. As all levels of memory are used, it is necessary to add the Roofline that represents each level. As shown in the example, the code performance exceeds the limit of DRAM and reaches the limit of the L2 cache. Including all data movement in the measurements allows for more straightforward performance prediction in a particular application code.

In order to prepare the Roofline model, two specific quantities are needed: W and Q. Section 5.1 describes different methodologies on how to obtain these quantities and addresses some of the limitations of the model. These new methodologies have been specifically developed and tested for the purpose of this study.

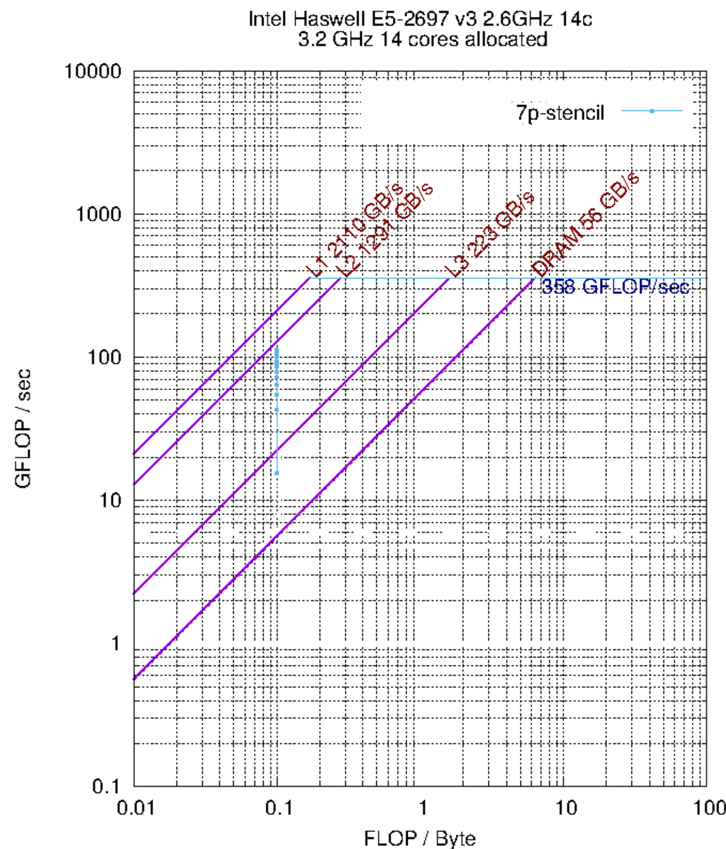


Figure 3.11: Cache-aware Roofline model of an example application

3.4 Methods to optimise energy consumption

There are many different approaches to improving energy efficiency of heterogeneous systems, including the commonly used dynamic voltage/frequency scaling (DVFS). Generally, in this approach the clock frequency is adjusted to reduce the supply voltage, and thus the power saving is achieved. For example, the authors proposed a technique to reduce energy usage by using multiple GPUs with each CPU instead of using a single CPU-GPU pair [16]. In this case, the CPU only manages the work of GPUs, and its frequency is reduced to save the energy further. Another approach is to adjust the CPU-GPU work distribution in order to improve energy efficiency. For instance, authors in [64] in the first step try to balance the load between the CPU and the GPU, so both processing units finish the computations

approximately at the same time. In the next step, they reduce the frequency of GPU components and the voltage of the CPU to achieve more significant energy savings with low-performance degradation. An interesting approach was developed in [63]. It is based on a linear programming model to distribute a workload between the CPU and the GPU using the profiled performance and the energy usage of the High Performance Linpack benchmark (see Section 2.1). Unlike our work, they do not take into account the communication cost. The other technique is based on a dynamic resource allocation of the processing units. A new method to predict the number of required cores of a processing unit to employ the power gating for the rest of the cores was proposed in [103]. The method constantly checks if the idle time is long enough to reduce the switching penalty. Another interesting technique is to apply application-specific optimisations. One example method to optimise the energy usage of stencil computations was introduced here [117]. The authors assumed that stencil computations might also be parallelised over time. This specific technique is called time tiling. This technique may be used only in stencil applications where there are no other computations within a single time step. This procedure changes a performance-limiting factor from memory-bound to compute-bound computations. The authors focused on minimising the off-chip memory accesses of a single CPU by improving the cache hit rate. Similarly, another research presented in [66] also focused on reducing the energy usage of stencil computations through temporal parallelisation. The authors analysed the influence of code optimisations on the arithmetic intensity of stencil computations and related it with the energy usage. A set of improvements to the energy usage by avoiding to use CPUs to communicate the data between distributed GPUs was presented in [75]. The proposed optimisation was achieved by employing dynamic parallelism of the GPU to handle the data transfer. It was demonstrated that after applying this optimisation the performance is slightly worse, whereas the performance per Watt increased significantly (up to 10%).

To the best of our knowledge, none of the previous research considered an energy-aware distribution of the stencil workload on heterogeneous computing resources with the time constraint. Moreover, none of them tried to minimise the energy consumption of intra-node and inter-node communications that significantly influence energy savings. These relevant assumptions have been considered in the conducted research.

3.5 Energy models

Recently, the Roofline model has been extended to take into account the energy consumption in GPUs [21]. In the new model, the authors have assumed that each operation has a fixed energy cost and a fixed data movement cost while the constant energy cost is linear in time. The constant power depends on the hardware

and an algorithm and includes static and leakage power management. However, the proposed model does not include dynamic power management. The dynamic power is affected by gate capacitance, supply voltage and operating frequency. DVFS changes the runtime supply voltage and the operating frequency, and it influences the dynamic power [40]. The authors assumed that the time per work (arithmetic) operation and the time per memory operation are estimated with the hardware peak throughput values, whereas the energy cost is estimated using a linear regression based on real experiments. Another set of extensions to the Roofline model has been proposed in [42] to model the energy on a dual multi-core CPU with the three-level cache hierarchy. In this approach, the dynamic power management was modelled as a second-degree polynomial, based on real benchmark data, and it scales linearly with the number of active cores up to the saturation point. The authors assumed that the dynamic power depends quadratically on the frequency. At the saturation point, the energy to solution grows with the number of used cores, which is proportional to the dynamic power, while the time to solution stays constant. Another example is an energy model presented in [104] to evaluate the cost of parallel algorithms for the GPU. Based on the energy model, the authors proposed a new method for energy scalability to ease the selection of the optimal number of blocks. In our work, two examples of architectures, CPUs and GPUs, are provided, see Section 5.3. However, our model can be utilised with other computing architectures, such as Intel Xeon Phi or ARM.

In our opinion, the energy can be modelled similarly to modelling the performance using the Roofline model. The Roofline energy model is also based on the notion of useful work and the number of moved bytes Q through the slowest data path. As for the Roofline model, the work is represented by the number of floating-point operations (FLOP) being executed W. This model assumes that calculating every $FLOP$ and sending any single byte of memory cost some energy. Moreover, during the execution of the code the processing unit also uses some power.

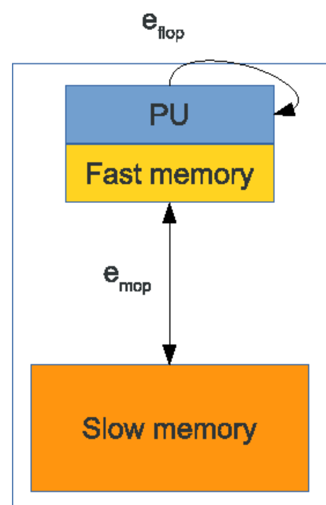


Figure 3.12: Roofline model of energy

The following equation is used to calculate the energy cost of application execution on a single processing unit:

$$E = W * e_{flop} + Q * e_{mop} + T * P0 \quad (3.6)$$

where e_{flop} is the energy cost per $FLOP$, e_{mop} is energy cost per single byte moved (MOP - memory operation) and $P0$ is the constant power drawn (see Figure 3.12). The constant power is based on a so called *P-state* of the processing unit and the number of cores used. The variables e_{flop}, e_{mop} and $P0$ are approximated with a linear regression. For the energy usage measurements, some hardware vendors expose API. For example, the RAPL interface is exposed for Intel CPUs, whereas Nvidia shares the NVML interface, see Section 2.3.2. The above presented model is extended for stencil computations in Section 5.3.

Chapter 4

Basic notions in the theory of algorithms and computational complexity

In this chapter, basic notions in the computational complexity of combinatorial problems are recalled due to [12, 36] where more details about the respective topics may be found. The description of three classes of combinatorial problems is presented. Two groups of algorithms based on computational complexity are described. This chapter also describes three examples of linear programs: ILP, MIP and BIP. The last section presents basic definitions from the graph theory and depicts an edge colouring problem.

4.1 Computational complexity

In general, combinatorial problems may be divided into three classes: search, decision and optimisation ones. These problems are defined by the finite or enumerative number of objects and parameters of an integer type. The combinatorial search problem for a given instance of the considered problem finds its feasible solution, otherwise no solution can be found. An example of the search problem is the eight queen problem [65]. In this problem, eight queens are placed on a chessboard in a way that any two queens can not be in the same row, column or diagonal. The decision problem provides only an answer with *yes* or *no* for a stated question. For instance, does the given graph contain the Hamiltonian cycle? The optimisation problems, together with the decision problems, are the subclasses of the search problems. The optimisation problem is formulated to minimise or maximise the objective function. The optimisation problem finds an optimal solution, otherwise no solution can be found. This problem may be formulated as the decision or search problem, but it can not be done in the opposite way. For example, the decision problem asks if there exists a solution with the objective value less than or equal to a given value in the case of the minimisation (larger or equal in the case of maximisation). For

some problems, the solution to the search problem may be equal to the solution of the optimisation problem, as the only feasible solution is the optimal solution.

Let π define the decision problem with a finite set of parameters. The values assigned to these parameters are called an instance which is denoted by I. D_π depicts the set of all instances for the problem π. The size of the instance I is denoted by $N(I)$. The instance I can be encoded by a finite string of symbols $x(I)$ where the symbols belong to the alphabet σ. It is assumed that the encoding is not represented by a string of 1 and is not redundant.

The algorithm A is a list of elementary steps, which solves the problem π for all its specific instances $I \in D_\pi$. A function of computational complexity $\eta_A(n)$ of the algorithm A that solves the problem π with a size $N(I) = n$ is a measure of the maximum number of elementary steps needed to solve the instance I of this size.

Definition 1 *If for a given $\eta_A(n)$ there exists a constant value $c > 0$ that $\eta_A(n) \leq c * f(N(I))$ for all $I \in D_\pi$ then $O(f(N(I))$ is a computational complexity of algorithm A.*

The algorithm can be further divided to two groups based on the computational complexity: polynomial and exponential. The polynomial algorithm is an algorithm with the computational complexity equal to $O(f(n))$ where f is a polynomial function $f(n) = a_0 + \ldots + a_{k-1} * n^{k-1} + a_k * n^k$ and $n = N(I)$. Any other algorithm where the computational complexity can not be defined as above is called exponential. The algorithm for which the polynomial algorithms are not known, or even the algorithm does not exist, is called intractable. In order to determine whether to use exact or heuristic algorithms, it must be known if the problem is intractable or not. The abstract model of the computing system is introduced to describe the classes of computational complexity. A well-known example is the Deterministic Turing Machine (DTM) which simulates a real computer system. An algorithm with a polynomial computational complexity on DTM is executed in polynomial time on any other realistic system. The Non-Deterministic Turing Machine (NDTM) represents the non-deterministic model where the number of executed instructions at a time is unbounded. The problem π with the polynomial complexity on NDTM is solved in $O(2^{p(N(I))})$ on DTM where p is a polynomial function and $I \in D_\pi$.

The definition of DTM and NDTM allows introducing the classes of computational complexity. The problem that is solved in the polynomial-time on DTM belongs to the P class (Polynomial). An NP class represents the problem where there is an algorithm that solves it in a polynomial time on NDTM. The well-known problem is that $P = NP$ is still open; however, most scientists expect $P \neq NP$. A polynomial transformation is introduced to define the next class of computational complexity.

Definition 2 *The polynomial transformation of the problem π_2 to the problem π_1 ($\pi_2 \propto \pi_1$) is called a function $f : D_{\pi_2} \rightarrow D_{\pi_1}$ that fulfils the following conditions:*

- *for any instance $I_2 \in D_{\pi_2}$ the answer is "yes" if and only if the answer for $f(I_2) \in D_{\pi_1}$ is "yes",*
- *the execution time of the function f on DTM for any instance $I_2 \in D_{\pi_2}$ is upper bounded by a polynomial $g(N(I_2))$.*

Definition 3 *Decision problem π_1 is NP-complete if it fulfils the following conditions:*

- $\pi_1 \in NP$,
- $\pi_2 \propto \pi_1$ *for each* $\pi_2 \in NP$.

It follows from the above definition that in order to prove NP-completeness of problem π_1 we have to first construct a polynomial-time algorithm on NDTM and prove that all problems from NP transform polynomially to π_1. The last class of the computational complexity is strongly NP-complete. The definition of a *pseudo-polynomial* algorithm is introduced to describe this class:

Definition 4 *The pseudo-polynomial algorithm is an algorithm where the function of computational complexity is upper-bounded by a polynomial depending on the instance size $N(I)$ and the maximum value of any problem parameter $Max(I)$.*

The strongly NP-complete class represents the decision problems that can not be solved in pseudo-polynomial for $P \neq NP$.

Definition 5 *Let π_p define a subproblem of decision problem π obtained by constraining D_π to instances $Max(I) \leq p(N(I))$ where p is polynomial. The decision problem π is strongly NP-complete if it belongs to NP, and there exists a polynomial p where π_p is NP-complete.*

A polynomial p that the subproblem π_p is NP-complete should be found to prove that the problem belongs to the strong NP-complete class. To simplify this proof one may utilise a pseudo-polynomial transformation:

Definition 6 *The pseudo-polynomial transformation of a problem π_2 to a problem π_1 is a function $f : D_{\pi_2} \rightarrow D_{\pi_1}$ where:*

- *for any instance $I_2 \in D_{\pi_2}$ the answer is "yes" if and only if for $f(I_2) \in D_{\pi_1}$ the answer is also "yes",*
- *execution time of the function f computed on DTM is upper-bounded by the following polynomials: $g_1(Max(I_2))$ and $g_2(N(I_2))$ for $I_2 \in D_{\pi_2}$,*
- *there exists a polynomial p_2 where for each $I_2 \in D_{\pi_2}$ such that: $Max(f(I_2)) \leq p_2(Max(I_2, N(I_2)))$.*

The problem π_1 is strongly NP-complete when there exists a pseudo-polynomial transformation of π_2 to π_1 and both π_2 is strongly NP-complete and $\pi_1 \in NP$.

Definition 7 *A polynomial Turing transformation of a search problem π_2 to a search problem π_1 is an algorithm A that solves the problem π_2 on DTM in a polynomial time and uses some hypothetical polynomial-time procedure S solving the problem π_1 on DTM.*

A search problem π_1 belongs to the NP-hard class if any other NP-hard problem π_2 may be transformed to π_1 using the polynomial Turing transformation $\pi_2 \propto \pi_1$.

Definition 8 *Let π_p define a subproblem of a problem π obtained by constraining D_π to instances $Max(I) \leq p(N(I))$ where p is polynomial. The decision problem π is strongly NP-hard if it belongs to NP, and there exists a polynomial p where π_p is NP-hard.*

An optimisation version of a problem belongs to the NP-hard class if its search version is NP-hard or its decision version is NP-complete. Therefore, the optimisation version of a problem is at least as complex as the search version, and the search version of the same problem is at least as complex as the decision version. The decision and search versions of a problem are not computationally harder than the optimisation version. However, the existence of the polynomial-time algorithm for the decision or search version of a problem does not determine the computational complexity of the optimisation version. Thus, to solve the NP-hard problem in a reasonable time one needs to develop heuristic algorithms at the expense of not finding the optimal solution.

4.2 Linear programming

This section provides a short overview of linear programming, and more details can be found in e.g., [38, 95, 89]. Before defining the linear program (LP), the basic concepts from linear algebra are introduced. A function F is a linear function with the following form:

$$F(x_1, x_2, \ldots, x_n) = a_1x_1 + a_2x_2 + \ldots + a_nx_n = \sum_{i=1}^{n} a_ix_i \tag{4.1}$$

where $x_1, x_2, \ldots, x_n$ are variables and $a_1, a_2, \ldots, a_n$ are some real numbers called *coefficients*. A linear equality is a following linear function F:

$$F(x_1, x_2, \ldots, x_n) = b \tag{4.2}$$

where b is a real number. Similarly, the linear inequalities are defined as:

$$F(x_1, x_2, \ldots, x_n) \leq b \tag{4.3}$$

$$F(x_1, x_2, \ldots, x_n) \geq b \tag{4.4}$$

The above linear equation and linear inequalities are called the linear constraints. The linear programming is defined with a minimisation or maximisation of a linear function subject to the set of the linear constrains. In a standard form the linear program is expressed as:

$$maximise/minimise \sum_{j=1}^{n} c_j x_j \tag{4.5}$$

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \; for \; i = 1, 2, \dots, m \tag{4.6}$$

$$x_j \geq 0 \; for \; j = 1, 2, \dots, m \tag{4.7}$$

where the result of the objective function is minimised or maximised subject to the $n+m$ constraints and all variables are real numbers. The compact form of the linear program is as follows:

$$maximise/minimise c^T x \tag{4.8}$$

$$Ax \leq b \tag{4.9}$$

$$x \geq 0 \tag{4.10}$$

where c^T is a row vector with n constants, x is a column vector with n variables, b is a vector with m constants and A is a matrix with n columns and m rows. The set of the x variables that satisfies all constraints is called a feasible solution; otherwise it is called an infeasible solution. An optimal solution is a maximum or a minimum value of the objective function for all feasible solutions for the maximisation or minimisation problem, respectively. The linear program is proved to be solved in a polynomial time [50]. However, in this study ILP is employed for which the polynomial time algorithm does not exist, assuming $P \neq NP$. In ILP the x variables are integral for all $j = 1, 2, \dots, n$, $x \in \mathbb{Z}$. If $x \in \mathbb{Z}^p \times \mathbb{R}^{n-p}$ for $p \in 1, \dots, n-1$ then it is called MIP. A special case where $x \in \{0, 1\}^n$ is called binary program (BIP).

4.3 Graph theory

4.3.1 Basic definitions

This section introduces some basic definitions from the graph theory [15]. An undirected graph G is an ordered pair $G = (V, E)$, where V is a finite set of elements called vertices, whereas E is a finite set of unordered pairs of vertices called edges. The cardinality of the set of vertices V is denoted by the symbol $n = |V|$ and called the order of graph G. The cardinality of the set of edges E is denoted by $m = |E|$ and called the size of graph G. A directed graph G^D is an ordered pair $G = (V, E^D)$ consisting of a finite set of elements called vertices V and a finite set of ordered

pairs of usually distinct vertices of G^D called directed edges E^D. An edge uv join the vertices u and v. In a directed graph, the edges in E^D are assumed to be ordered pairs and are described as $(u, v) \in E^D$. An ordered pair (u, v) is an edge directed from u to v.

4.3.2 Multiplicity

Two edges $\{uv\}, \{st\} \in E$ are parallel if $\{u, v\} = \{s, t\}$. The multiplicity of an edge $\{uv\} \in E$ is the number of edges parallel to uv:

$$\mu_{uv} = |\{st \in E : \{uv\} = \{st\}\}| \tag{4.11}$$

4.3.3 Adjacency and incidence

Two edges $\{uv\}, \{st\} \in E$ are adjacent if $\{u, v\} \cap \{s, t\} \neq \emptyset$ and edge $\{uv\} \in E$ is called incident to its both end vertices u and v. The set of edges incident at a vertex u is denoted by $\delta_G(u)$:

$$\delta_G(u) = |\{e \in E : \{e\} \cap \{u\} \neq \emptyset\}| \tag{4.12}$$

The number of edges incident to a vertex u is the degree of this vertex in G and will be denoted by $deg_G(u)$. For $U \subseteq V$, the set of all edges with exactly one endpoint in U is denoted by $\delta(U)$. For a vertex $u \in V$ in a directed graph $G^D = (V, E^D)$ we define $\delta^+_{G^D}(u) : \{(v, w) \in E^D : v = u\}$ as the set of edges leaving the vertex u and $\delta^-_{G^D}(u) : \{(v, w) \in E^D : w = u\}$ as the set of edges entering the vertex u. The vertices u and v are called adjacent if $\{u, v\} \in E$.

4.3.4 Maximum degree and maximum multiplicity

The maximum degree and maximum multiplicity of a graph are defined as

$$\Delta G = \max_{v \in V} deg_G(v) \tag{4.13}$$

$$\mu G = \max_{e \in E} \mu_G(e) \tag{4.14}$$

A graph with $\mu(G) = 1$ that contains no parallel edges is called simple. Graphs with maximum multiplicity of at least 2 are called multigraphs and denoted by M.

4.3.5 Edge colouring

Edge colouring of a graph $G = (V, E)$ is a map $c : E \to C$ which assigns to each edge $e \in E$ a colour $c(e) \in C$ such that no two adjacent edges receive the same colour.

The minimal cardinality of the colour set C for which such a mapping exists is called the chromatic index of the graph and denoted by $\chi'(G)$. The edge colouring problem is proved to be an NP-complete problem[46]. To prove the NP-completeness, the edge colouring problem is restricted to a decision problem whether the chromatic index of a cubic graph is equal to three or four. A graph where all vertices have a degree equal to three is called a cubic graph. The authors utilise a polynomial transformation from the known NP-complete problem 3-SAT (3 boolean satisfiability). 3-SAT problem is represented in a conjunctive normal form where each clause is limited to three literals. However, there are strong lower and upper bounds for the chromatic index. From the definition of edge colouring, the lower bound form the chromatic index is $\Delta(G) \geq \chi'(G)$. The upper bound for the chromatic index proved by Vizing [102] is $\chi'(G) \geq \Delta(G) + 1$. The chromatic index is determined by the degree of the graph and can take only two values: $\Delta(G)$ or $\Delta(G) + 1$. Still, the calculation of the chromatic index of a graph remains the NP-hard problem. Thus, there exist different suboptimal approaches that try to address this problem. One of the well-known methods is Vizing's algorithm that never uses more than $\chi'(G) + 1$ colours. The edge colouring problem is utilised in the problem formulation described in Section 5.2.

Chapter 5

Energy-aware resource management of stencil computations

In this chapter, we describe computational experiments that enabled us to discover the critical parameters that impact the performance and energy usage of stencil computations. We formulate our problem and define the performance and energy model. We also present a method based on ILP to obtain the optimal solution for the formulated problem.

5.1 Designation of the parameters

We have run several computational experiments to discover critical parameters that have a relevant impact on the performance and energy usage of a stencil task running on a particular processing unit. Thanks to the dynamic power management policies introduced on PUs, both the frequency and the number of cores have been controlled during our performance tests. The RAPL and NVML interfaces are used during the computational experiments. The RAPL interface provides the ability to monitor and control the power on the CPU socket and DRAM, whereas the NVML interface allows the management of the power states of the GPU. On multi-core CPUs, the frequency switching implicitly changes an operating voltage. The voltage is optimised by the hardware based on several factors. All processor cores that execute some workload share the same frequency and voltage. The multiple frequency and voltage pairs while executing code are called P-states. The idle cores switch to low-power idle states called C-states. At the higher levels of C-states power save actions are taken such as flush of the caches, stop of the clocks and reduction of the voltage to zero.

Figure 5.1 shows the performance of a seven-point stencil on eight core Intel Xeon E5 CPU and Kepler K20m GPU using different P-states. In case of the CPU the maximum performance can be reached with four to six cores depending on the frequency used. For the GPU all available cores execute workload and the maximum performance is achieved with the 705 MHz clock. The stencil computations are a

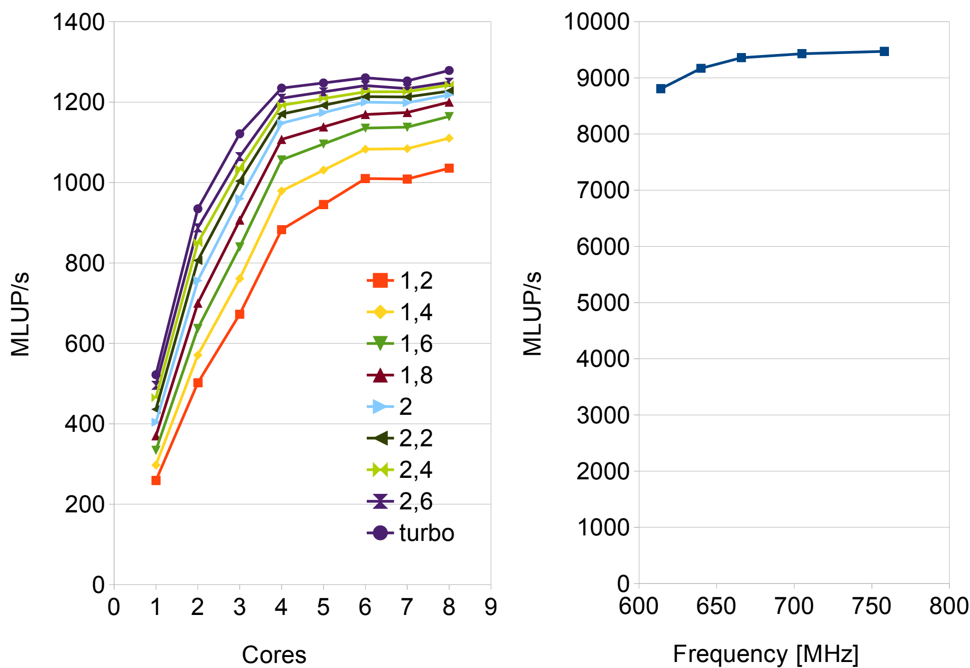


Figure 5.1: Performance of a seven-point stencil: left - the CPU, right - the GPU

Table 5.1: Properties of the modern architectures. The GPU bandwidth with Error Correcting Code (ECC) switched on/off.

Platform	CPU Xeon E5-2670@2.60GHz	GPU Kepler K20m
Peak perf.	173 Gflop/s	1168 Gflop/s
Bandwidth	30 GB/s	143/173 GB/s
Ratio	5.76	8.17/6.75

class of memory-bound problems as the stencil's theoretical flop to byte ratio, which is typically less than 0.5, is significantly lower than that of the current computing architectures, see Table 5.1. In practice, the flop to byte ratio can be even lower due to a blocking overhead, as each stencil has to be divided to separate blocks to be effectively computed. The sizes of the blocks were carefully selected to fit in a cache hierarchy of the target PU and to minimise the memory bandwidth pressure. Due to the memory bottleneck, the performance saturation is reached with a reduced number of cores. The four cores clocked at the turbo frequency are needed to saturate the memory bandwidth, whereas for the higher number of cores the lower frequency is needed, see Figure 5.2. The comparison of Figures 5.1 and 5.2 shows that there is a strong relationship between the memory bandwidth and the stencil performance.

Figure 5.3 shows an influence of the P-states on energy usage for a seven-point

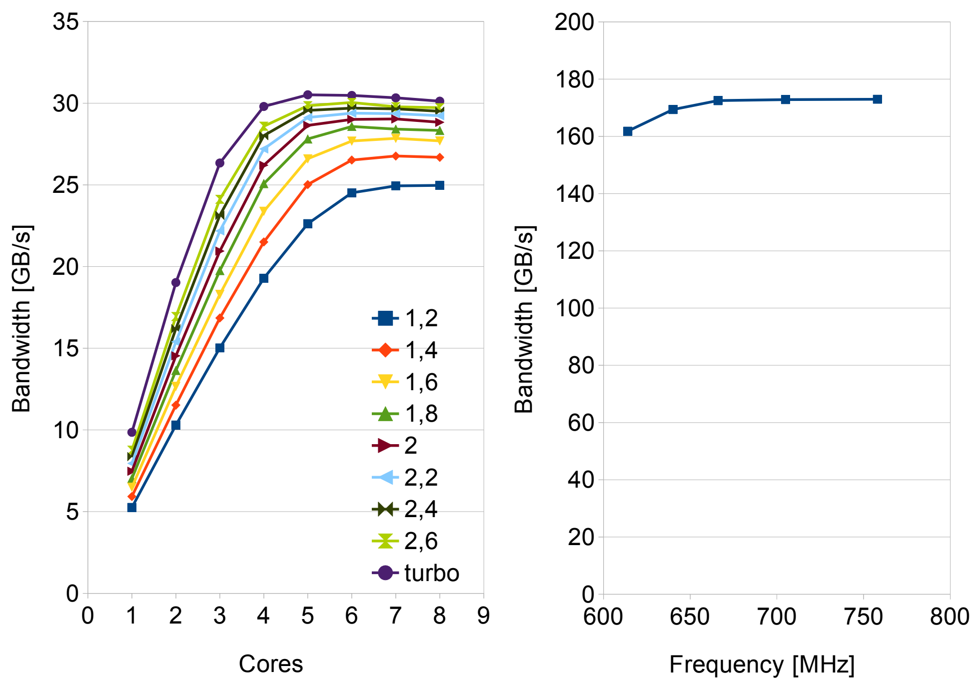


Figure 5.2: Memory bandwidth benchmark on: left - the CPU, right - the GPU

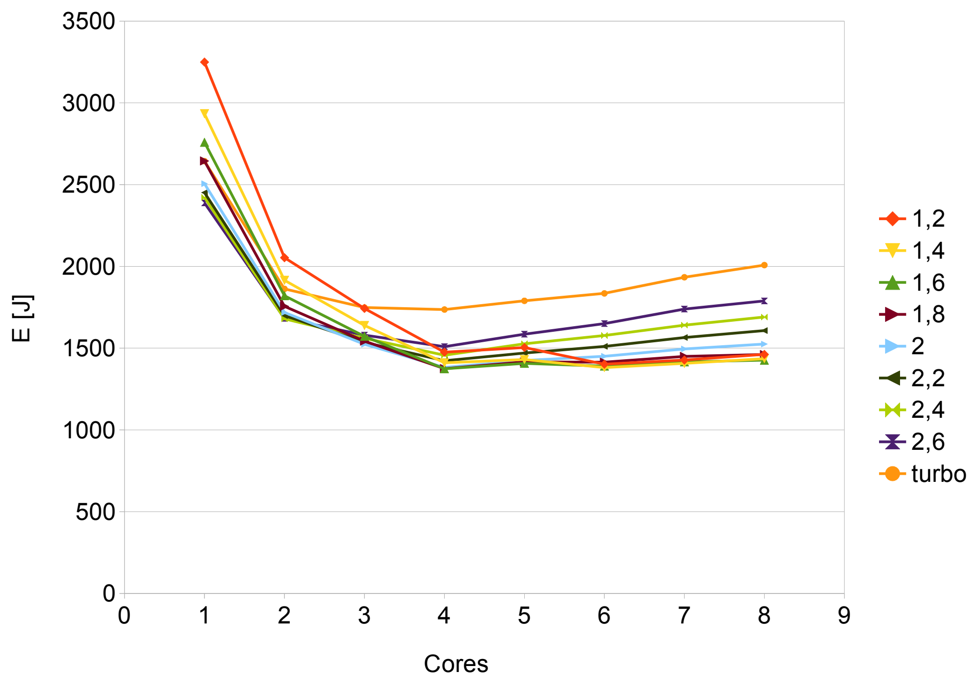


Figure 5.3: Energy usage of a seven-point stencil on the CPU

stencil. The lowest energy usage is reached with four cores clocked at 1.6 GHz. From four to eight cores, the difference in the energy usage is only 4% for the optimal

frequencies; thus, to save energy while increasing the number of cores the frequency should be minimised. The lowest energy usage is reached not with the maximum performance of 1278 MLUP/s, but with the lower performance of 1107 MLUP/s. This is contrary to expectations that computations with the highest performance are most energy-efficient. The power consumption of the modern CPU consists of two parts: static and dynamic power. The static power is referred to as a leakage power where the power is lost due to a current that finds its way to the ground. This process happens regardless of the operating frequency, and it depends entirely on the voltage.

$$P_{static} = m * V_{cc} \tag{5.1}$$

The m value is a constant coefficient, and V_{cc} is the CPU voltage. The dynamic power involves two additional terms: one referred to as a short-circuit energy, and the other one is called transition energy:

$$P_{transition} = \alpha * C * f/2 * V_{cc}^B \tag{5.2}$$

$$P_{short-circut} = \alpha * E_{short-circut} * f \tag{5.3}$$

$$P_{dynamic} = P_{shot-circut} + P_{transition} \tag{5.4}$$

where α is the activity parameter that depends on the instruction mix and overall instruction per cycle (IPC) utilization, the C parameter is dependent on the layout of the chip, the f parameter is a current frequency, while the $E_{short-circut}$ parameter is the power consumed per short-circuit event.

Figure 5.4 presents the power consumption of a seven-point stencil. Based on Eq. 5.4 the theoretical relation of $P \sim V^B$ is not reflected in the figure as the power consumption changes linearly with the increasing voltage. Furthermore, the power consumption of the three most essential CPU components was checked, including package (PKG), cores (PP0) and DRAM. The PKG component includes power consumption of the whole processor die, whereas PP0 only contains power consumption of the cores.

Figure 5.5 shows that for the 1.2GHz clock, 60% of the power consumes DRAM, whereas for the 2.6GHz clock form 50% to 60% of the power consumes PKG. The power consumption of PP0 changes linearly with the increasing number of cores.

Figure 5.6 depicts power consumption based on the lowest energy usage for each P-state. In our case, data movement consumes most of the power, as the PKG part also includes power consumption of the caches. Figure 5.7 presents that the computation of a seven-point stencil on the GPU is 10x more energy and performance efficient than on the CPU.

To summarise the analysis, the maximum performance can be achieved with fewer cores than available. Secondly, it is more important to reduce the frequency than the number of cores used to minimise energy usage. What is more, in the case of the CPU, DRAM may use up to 60% of the energy. Thus, the data movement

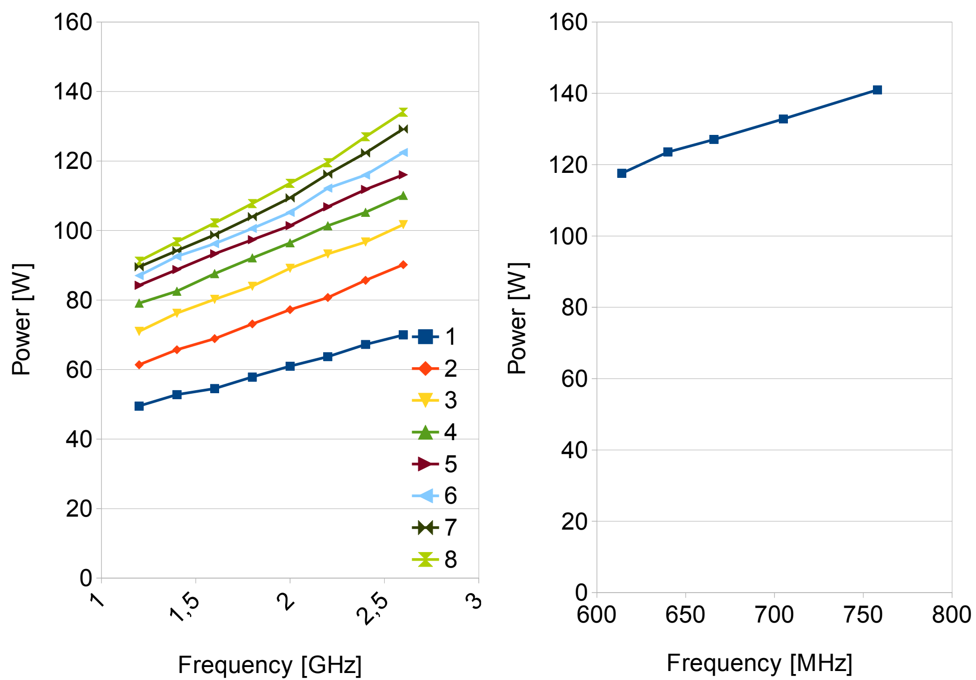


Figure 5.4: Power consumption of a seven-point stencil: left - the CPU, right - the GPU

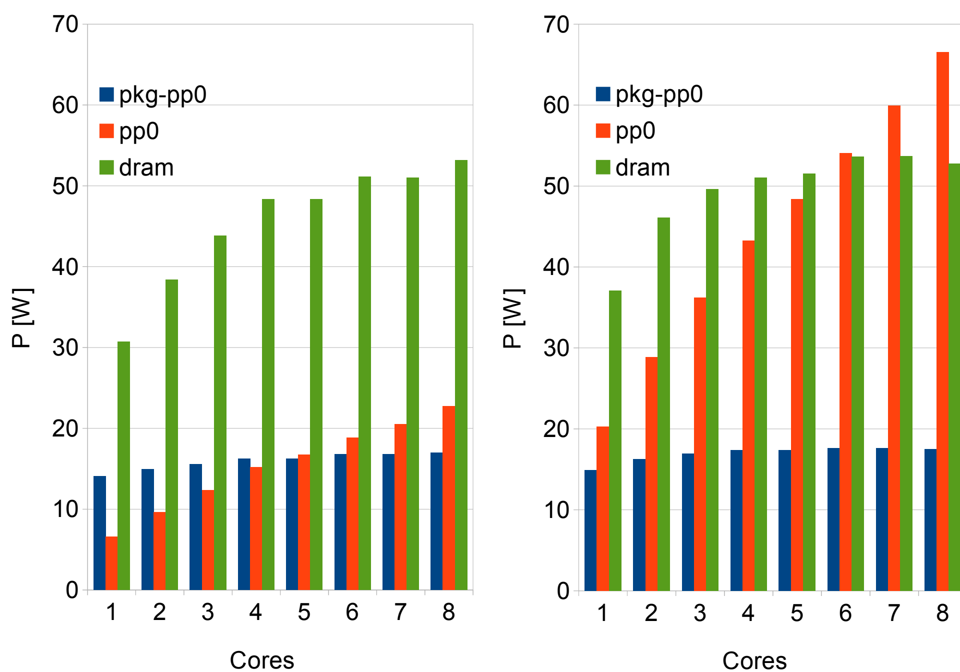


Figure 5.5: PKG, core and DRAM power consumption of a seven-point stencil on the CPU

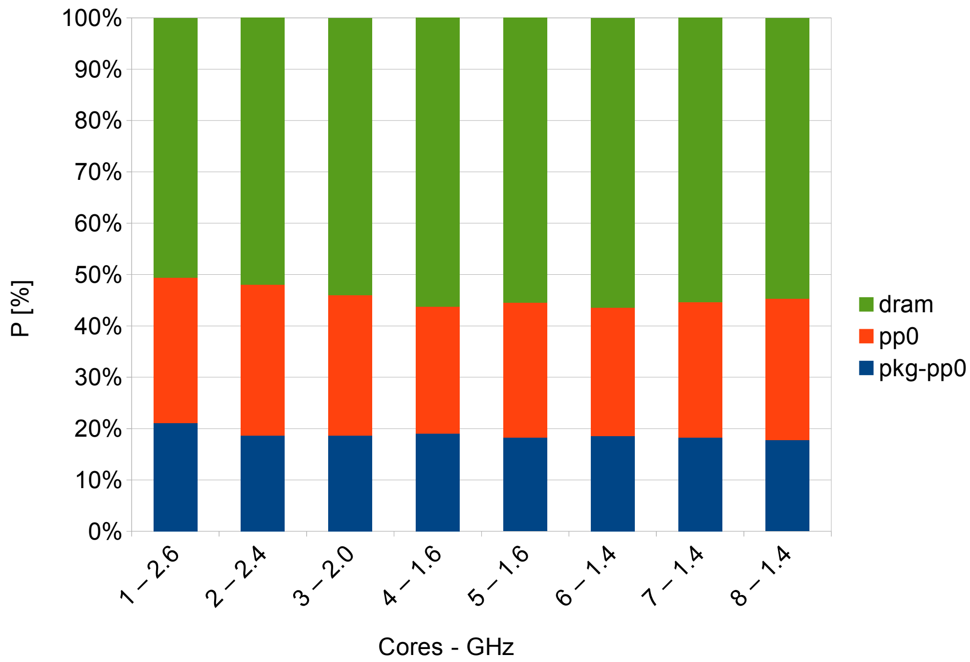


Figure 5.6: Power consumption of a seven-point stencil on the CPU

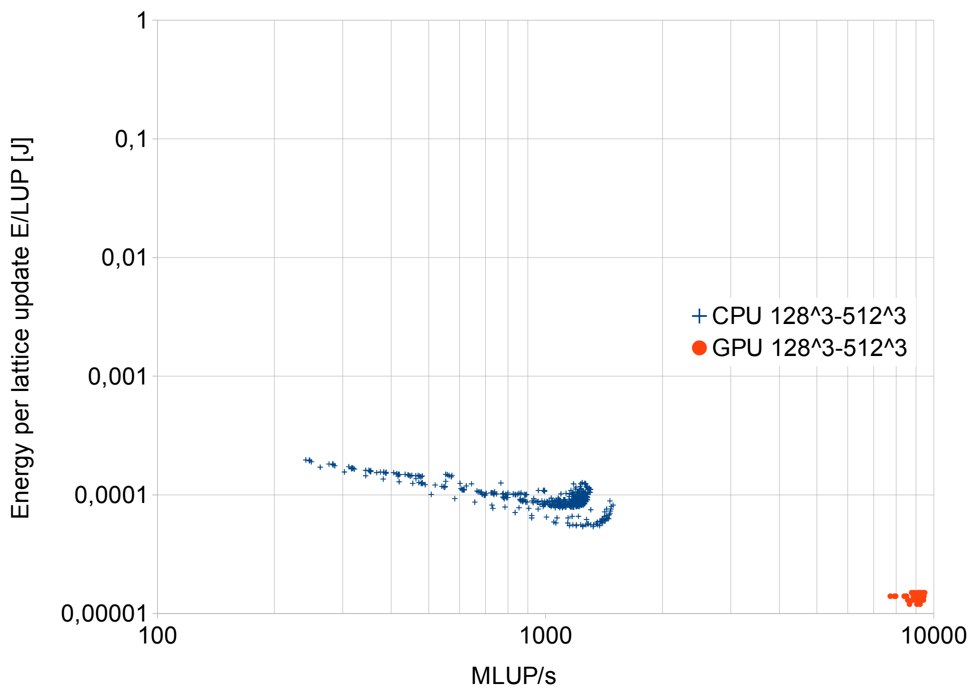


Figure 5.7: Performance to energy cost per lattice update for a seven-point stencil using different domain sizes on the CPU and the GPU

consumes most of the power. Finally, the lowest energy usage may be reached with not the maximum performance.

5.2 Problem formulation

As showed in the previous section that the data locality has the most substantial influence on energy usage, it has encouraged us to focus our research on a stencil workload scheduling using heterogeneous computing architectures to minimise the energy usage while meeting the computation deadline.

In general, the stencil problem that we consider is defined on a structural grid which consists of grid cells defined in three dimensions. A single update of the whole grid is called a timestep. Each cell in the grid is updated with a strict pattern that defines which neighbouring cells are used during a stencil computation, see Figure 5.8. We have assumed that the stencil communication pattern is static. This assumption was based on our experience with many real stencil simulations in HPC setups, where the number of parallel processes defined in the initial mapping of parallel stencil processes onto heterogeneous processors remains constant during their execution. In our approach we have focused on an explicit method where a

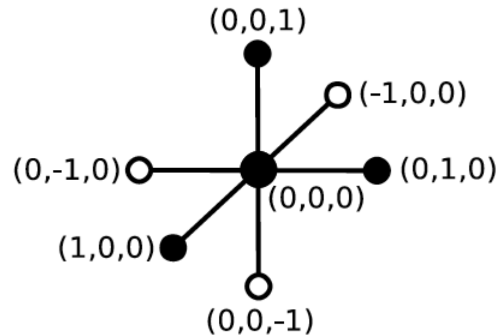


Figure 5.8: seven-point reference stencil communication pattern.

present timestep is updated by values of the grid cells from a previous timestep. We have distinguished two main classes of unrelated processors commonly used in HPC setups: CPU and GPU processors, as it was proposed in [26]. We have assumed that the same stencil computation may have different runtime values executing on heterogeneous and unrelated processors. The block decomposition of the structural grid updated by the stencil forms the workload of tasks with the communication dependencies. Each task represents a single block of the decomposed grid. The grid is decomposed on equally sized blocks. A given task may be processed by a single processor at a time, and each processor may execute several tasks. The communication is executed in parallel between different pairs of processors. However, each processor can initiate a single communication link with another processor at a given time. As a result, we must employ several communication rounds to exchange all

data. Naturally, the number of communication rounds directly influences the overall communication time. The communication and the computation are done in parallel. Typically, in stencil computation implementations, a lot of the effort is put into the parallel exchange of the grid cells between processors and the computations on the processors.

Based on the assumptions mentioned above, we formulate our problem as a complete directed graph $K = (V^P, E^P)$ of m unrelated processors and a directed graph $G = (V^T, E^T)$ of n dependent tasks. The problem is to minimise the energy usage by mapping a graph G on a graph K for a given deadline t^d under the following assumptions:

1. $t^s < t^d$, where t^s is the total execution time of stencil computations,
2. $max(t^e_{T_u,P_i,L_a}) * d_{T_u} < t^s$ for each $T_u \in \mathcal{V}^\mathcal{T}$, for each $P_i \in \mathcal{V}^\mathcal{P}$ and for selected $L_a \in \mathcal{L}$,
3. $d_{T_u} * Q_{T_u,P_i} < r_{P_i}$ for each T_u mapped on processor P_i,
4. $z * GCD(\underset{(P_i,P_j)\in E^P}{\forall} t^c_{(P_i,P_j)}) * C_{(T_u,T_v)} < t^d$, where z is the number of communication rounds (colours), GCD is the greatest common divisor, $t^c_{(P_i,P_j)}$ is the communication time and $C_{(T_u,T_v)}$ is the number of grid cells to communicate.

The scheduling model assumes a fully connected network of heterogeneous processors with different communication topologies. The directed graph $K = (V^P, E^P)$ represents a network topology with two types of communication links: intra-node and inter-node. It is described by the following parameters:

1. V^P the set of processors
2. P_i the processor i
3. $P^{idle}_{P_i}$ the power usage in an idle state when no computations are executed
4. r_{P_i} the processor memory size
5. E^P the set of edges
6. (P_i, P_j) the communication link (edge) between processors P_i and P_j
7. $t^c_{(P_i,P_j)}$ the communication time between processors P_i and P_j.

Each processor P_i is characterised by the following parameters:

1. $\mathcal{C} = \{c_1, c_2, \ldots, c_h\}$ the set of available cores
2. $\mathcal{F} = \{f_1, f_2, \ldots, f_g\}$ the set of available frequencies
3. $\mathcal{L} = \{(f, c) : f \in \mathcal{F} \wedge c \in \mathcal{C}\}$ the set of states where $L_a \in \mathcal{L}$ is a selected state
4. b_{P_i,L_a} the sustained bandwidth to the processor memory in bytes per second
5. $P^{perf}_{P_i,L_a}$ the performance in the floating-point operations (double precision) per second.

Tasks are represented by the directed graph $G = (V^T, E^T)$, where V^T denotes the set of tasks and E^T represents the set of edges. Each edge $(T_u, T_v) \in E^T$ defines the communication between the tasks $T_u, T_v \in V^T$. Graph $G = (V^T, E^T)$ is described by the following parameters:

1. V^T the set of tasks
2. T_u the task u
3. d_{T_u} the number of grid cells to process
4. E^T the set of edges
5. (T_u, T_v) the edge between vertices T_u and T_v
6. $C_{(T_u,T_v)}$ the number of grid cells to communicate.

Each task T_u is described by the following parameters in relation to processor P_i:

1. W_{T_u,P_i} the number of arithmetic operations in double precision per grid cell
2. Q_{T_u,P_i} the number of required bytes to update a grid cell
3. $t^e_{T_u,P_i,L_a} = max(O_{T_u}/P^{perf}_{P_i,L_a}; B_{T_u,P_i}/b_{P_i,L_a})$ the time to execute the single stencil task T_u, where O_{T_u,P_i} is the number of arithmetic operations to update the stencil task T_u and B_{T_u,P_i} is the number of bytes to update the stencil task T_u, see (5.8)
4. $P^0_{T_u,P_i,L_a}$ the constant power usage
5. $e^e_{T_u,P_i,L_a}$ the energy usage to execute the single stencil task T_u, see (5.9)
6. $t^c_{(T_u,T_v),(P_i,P_j)}$ the exchange time of moving task data between processors
7. $e^c_{(T_u,T_v),(P_i,P_j)}$ the energy cost of moving task data between processors.

A more detailed description of the parameters mentioned above is presented in our previous work [26]. The objective is to determine a schedule such that the total energy cost is minimised and deadline t^d is not exceeded, defined as follows:

$$min \sum_{P_i \in V^P} (e^e_{T_u,T_p,L_a} * A_{P_i} + t^{idle}_{P_i} * P^{idle}_{P_i}) + e^c \tag{5.5}$$

where $t^s \leq t^d$, A_{P_i} is the number of stencil tasks assigned to the processor P_i and $t^{idle}_{P_i}$ is the idle time where the processor P_i does not execute any computations.

5.3 Performance and energy model

A detailed analysis of the performance and energy usage of stencil computations on two unrelated processing units resulted in the following formulation of the performance model.

Computation time $t^c_{T_u,P_i,L_a}$ of task T_u on processor P_i with state L_a is estimated as follows:

$$O_{T_u,P_i} = W_{T_u,P_i} * d_{T_u} \tag{5.6}$$

$$B_{T_u,P_i} = Q_{T_u,P_i} * d_{T_u} \tag{5.7}$$

$$t^e_{T_u,P_i,L_a} = max(O_{T_u}/P^{perf}_{P_i,L_a}; B_{T_u,P_i}/b_{P_i,L_a}) \tag{5.8}$$

where $O_{u,p}$ is the number of arithmetic operations executed and $B_{u,p}$ is the number of bytes transferred.

The energy model assumes that each arithmetic operation and memory operation consumes some energy:

$$e^e_{T_u,P_i,L_a} = e^{op}_{T_u,P_i,L_a} * O_{T_u,P_i} + e^{byte}_{T_u,P_i,L_a} * B_{T_u,P_i} + P^0_{T_u,P_i,L_a} * t^e_{T_u,P_i,L_a} \tag{5.9}$$

Variables $e^{op}_{T_u,P_i,L_a}$, $e^{byte}_{T_u,P_i,L_a}$ approximate the energy usage of stencil operations. For simplicity, it is assumed that arithmetic operations, i.e., additions, multiplications, subtractions and divisions, consume the same amount of energy. Additionally, the energy usage also depends on an instruction set used; thus, the CPU implementation of the stencil computation uses vector extensions for the highest performance. $P^0_{T_u,P_i,L_a}$ is a constant power consumed by the processor P_i based on the state L_a.

5.4 Time measurement

This section aims to present a procedure to obtain stencil parameters on different processing units, including CPUs and GPUs. The benchmarking data collected during the preparation of the performance model and the energy model serves as the ground truth to validate the model. In order to prepare the model, two specific quantities are needed: W and Q. The following sections describe different methodologies on how to obtain these quantities. These methodologies have been specifically developed and tested for the purpose of this study.

5.4.1 Code analysis

The W_{T_u,P_i} and Q_{T_u,P_i} quantities can be obtained from analysing the source code of the seven-point stencil computation applied to the grid of a size 256^3, see Algorithm 1.

The analysis can be conducted based on the assumption of efficiency of the underlying memory hierarchy. The worst case assumes that none of the data can be cached during the execution of the stencil statement in lines 5 and 6. The best-case scenario assumes the infinitely large cache where all data can be cached. Firstly, the analysis is provided for the CPU. For the worst case (WC), where a single

Algorithm 1 Seven-point stencil

1: **procedure** SEVEN-POINT STENCIL
   Input: A grid D_t and a constant θ.
   Output: A grid D_{t+1}.
2: **for** k $=, ..., 256$ **do**
3: **for** j $=, ..., 256$ **do**
4: **for** i $=, ..., 256$ **do**
5: $D_{t+1}(i,j,k) = \theta * (D_t(i,j,k) + D_t(i+1,j,k) + D_t(i-1,j,k)+$
6: $D_t(i,j+1,k) + D_t(i,j-1,k) + D_t(i,j,k+1) + D_t(i,j,k-1))$
7: **end for**
8: **end for**
9: **end for**
10: **end procedure**

grid cell is processed, seven FLOPs are executed, eight words are loaded, and one word is stored. There are two types of operations: a single multiplication and six additions. Each word is the size of eight bytes as the double-precision computations are typically employed in scientific applications. seven words are loaded on the right-hand side of the statement, whereas on the left-hand side, a single word is loaded before storing the result. In cache-coherent architectures like Intel CPUs, each store to the memory is preceded by a cache line load from memory. It is so called write allocate, see Figure 5.9.

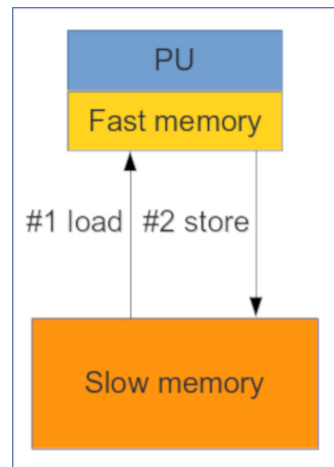


Figure 5.9: Write allocate in Intel CPUs

For the best case (BC), seven FLOPs are executed, one word is loaded and one word is stored. The write allocate can be omitted with the streaming stores. However, they are not always appropriate, as they omit cache if the following computations reuse the results of the computations. In summary, W_{T_u,P_i} and Q_{T_u,P_i} for an update of a single grid cell are equal to:

$$W^{WC}_{T_u,P_{CPU}} = 7\ FLOP \tag{5.10}$$

$$W^{BC}_{T_u,P_{CPU}} = 7\ FLOP \tag{5.11}$$

$$Q^{WC}_{T_u,P_{CPU}} = 9 * 8 = 72\ Bytes \tag{5.12}$$

$$Q^{BC}_{T_u,P_{CPU}} = 2 * 8 = 16\ Bytes \tag{5.13}$$

where WC means the task with the worst-case estimation and BC is the task with the best-case estimation. In the case of the GPU, the same methodology applies, and the only difference is for the worst-case scenario where there is no write allocate:

$$W^{WC}_{T_u,P_{GPU}} = 7\ FLOP \tag{5.14}$$

$$W^{BC}_{T_u,P_{GPU}} = 7\ FLOP \tag{5.15}$$

$$Q^{WC}_{T_u,P_{GPU}} = 8 * 8 = 64\ Bytes \tag{5.16}$$

$$Q^{BC}_{T_u,P_{GPU}} = 2 * 8 = 16\ Bytes \tag{5.17}$$

Next, as described in Section 5.3, the performance and bandwidth have to be designated to calculate the execution time. The performance is calculated based on the specification of the processor, whereas the bandwidth is measured by using a benchmark. In the analysis, the following processing units are used:

- CPU Intel Sandy Bridge Xeon E5-2670@2.6GHz
- GPU Nvidia Kepler K20m@705MHz.

The Intel Xeon E5-2670 CPU has eight cores clocked at 2.6 GHz where each core is able to compute two double-precision operations per cycle. The resulting performance is 166 GFLOP/s. However, this performance is limited by the load and store throughput of the AVX extensions, which are able to execute 1 AVX load and 1/2 AVX store per cycle. Thus, instead of calculating 8 FLOP per cycle, it requires three cycles per 8 FLOP. With this limitation, the performance is equal to 55.3 GFLOP/s. The Likwid tool is used [97] to measure bandwidth. The bandwidth with the CPU clocked at 2.6 GHz is equal to 30 GB/s. Nvidia K20m GPU includes 13 SMX multiprocessors clocked at 705 MHz where each has 64 DP units capable of executing two double-precision operations per cycle. The performance of this GPU is equal to 1170 GFLOP/s. The bandwidth measured with a benchmark is equal to 173 GB/s with Error Correcting Code (ECC) off. The execution time on CPU and GPU processors using the developed performance model for a stencil task T_u with the number of grid cells d_{T_u} equal to 256^3 is presented in Table 5.2.

As shown in Table 5.2, the CPU can achieve 5.4% and 24.3% of its peak performance for the worst- and best-case scenarios, respectively. The GPU can reach 1.6% and 6.47% of its peak performance for the worst- and best-case scenarios, respectively. The code analysis methods allow estimation of the parameters needed to calculate the execution time of the stencil computation without the need to access and execute the code on the processing unit. The disadvantage of this method is

Table 5.2: Execution time for the CPU and GPU architectures based on the code analysis.

Parameter	WC		BC	
	P_{CPU}	P_{GPU}	P_{CPU}	P_{GPU}
O_{T_U,P_i} [MFLOP]	112	112	112	112
B_{T_U,P_i} [MB]	1152	1024	256	256
$t^e_{T_u,P_i,L_a}$ [ms]	max(1.98, 37.5)= 37.5	max(0.09, 5.78)= 5.78	max(1.98, 8.33)= 8.33	max(0.09, 1.45)= 1.48

the lowest precision comparing the methods described in the following two sections.

5.4.2 Hardware performance counters

In this section another method utilizing Hardware Performance Counters is used to depict the W_{T_u,P_i} and Q_{T_u,P_i} quantities. Modern architectures include hardware support to monitor microprocessor activity. For example, Intel developed Intel Performance Monitoring Unit (PMU) that enables measuring different traits such as instruction retired, elapsed core clock ticks, L2/L3 cache hits and misses, as well as core and uncore events. Different open-source tools enable access to Hardware Performance Counters such as Intel Performance Counter Monitor (PCM), Likwid or perf. In this study, the Likwid tool is used as it supports CPUs from two main vendors: Intel and AMD, and because it does not need the root privileges to read the performance counters values. In order to measure the seven-point stencil implemented with a method described in Section 3.2.1, the wrapper functions must be added to the source code. Likwid has a marker API that measures selected regions of the code.

After the seven-point stencil function is wrapped with Likwid marker API calls, the Likwid tool is used.

For this example, different compilers with the following flags are used to check the differences in measured quantities:

- gcc 4.8.5 with "-O2 -march-core-avx2" or "-O3 -march-core-avx2";
- gcc 5.4 with "-O2 -march-core-avx2" or "-O3 -march-core-avx2";
- icc 16.0.3 with "-O2 -xCORE-AVX2" or "-O3 -xCORE-AVX2".

The processing unit used in the tests is Intel Xeon E5-2697 v3 with 14 cores clocked at 2.6 GHz with DDR4 clocked at 2133 MHz based on the Haswell architecture. This processing unit has a vector size equal to 256 bits (AVX2) and can fit 4 double-precision values at once.

Table 5.3 shows the obtained W_{T_u,P_i} for different compilers and flags.

Both versions of GCC for the O2 flags give the number AVX instructions equal to 0. The assembly code shows that GCC for this flag multiply-adds scalar double-precision values (vfmadd213sd). GCC with O3 flag and ICC multiply-adds vector/-

Table 5.3: W_{T_u,P_i} for different compilers and flags on the CPU.

Compiler flag	gcc 4.8.5	gcc 5.4	icc 16.0.4
O2 [MFLOP]	0	0	142.24
O3 [MFLOP]	142.24	142.24	142.24

packed double-precision values (vfmadd231pd). This example shows the importance of selecting appropriate compilation flags.

Only the AVX instructions are measured because there are no Hardware Performance Events for other floating-point operations on the Haswell platforms. This is the first issue with this method of obtaining the number of floating-point operations W_{T_u,P_i}. The second issue is that the measurements for FLOP are overcounted up to 35% [105]. In the case of Intel CPU, to obtain the more precise results of W_{T_u,P_i} the instrumentation is needed.

Table 5.4 shows the obtained Q for different compilers and flags.

Table 5.4: Q_{T_u,P_i} for different compilers and flags on the CPU.

Compiler flag	gcc 4.8.5	gcc 5.4	icc 16.0.4
O2 [MB]	640	640	512/640
O3 [MB]	640	640	512/640

For GCC, regardless of the used flags, Q_{T_u,P_i} is equal to 640 GB. ICC by default uses a heuristic to determine whether the write allocate is needed, and for the seven-point stencil routine Q_{T_u,P_i} equals 512GB. When a heuristic is explicitly switched off through compiler flags, Q_{T_u,P_i} equals 640 GB. With the use of Hardware Performance Counters, this method allows estimating the amount of data moved through the memory hierarchy with a reasonable precision. However, it is not precise or even does not allow estimating the number of executed floating-point operations. The last method presented in the next section is most precise among the already described methods.

5.4.3 Instrumentation

Instrumentation must be used to address the problem of counting the number of FLOPs in Hardware Performance Counters for Intel CPU. For this purpose, Intel Software Development Emulator (SDE) is used. SDE is a user-level function that emulates instructions of the Intel64 instruction set. This tool enables generating histograms of instructions executed, their length and category. For example, the number of scalar and vector instructions may be counted.

Table 5.5 presents the generated histogram for ICC compiler. Every floating-point instruction given in the histogram must be decoded. In this example, the VFMADD213PD_YMMqq_YMMqq_MEMqq instruction is executed 250M times

Table 5.5: Generated histogram for ICC compiler.

Instruction	No.
VFMADD213PD_YMMqq_YMMqq_MEMqq	250000000
VMOVSD_XMMdq_MEMq	10
VMOVUPD_MEMqq_YMMqq	250000000
VMOVUPD_YMMqq_MEMqq	250000000

where P is a packed instruction (vector), D is a double-precision data type, YMM is a register of a length equal to 256 bits, and FMADD is fused multiply-add (two floating-point operations), thus W_{T_u,P_i} equals 112 MFLOP. Both GCC versions with O2 flag produces histogram with the VFMADD213SD_XMMdq_XMMq_MEMq instruction. In this case, the instruction is a scalar with a register size equal to 128 bits that use fused multiply-add. Q_{T_u,P_i} also equals 112 GFLOP. The instrumentation allows counting the floating-point operations of any type.

The instrumentation that involves the actual execution of the code with careful tracking of the execution of each instruction is the most precise method to estimate the number of the executed floating-point operations. However, this method significantly slows down the execution time of the measured code even up to 30x times.

5.5 Energy measurement

The coefficients $e^{op}_{T_u,P_i,L_a}$, $e^{byte}_{T_u,P_i,L_a}$ and $P^0_{T_u,P_i,L_a}$ are approximated with a linear regression. Table 5.6 shows estimated values of the energy cost for the double-precision floating-point operation and the transfer of a single byte of data. For the CPU and GPU, the cost to transfer a single byte of data is 5.2x and 6x more expensive than the floating-point operation, respectively. What is more, both floating-point and memory operations are 5x more expensive on the CPU than on the GPU. Figure 5.10 shows that the constant power grows linearly with the increasing number of cores using different P-states.

Table 5.6: Energy coefficients for the CPU and GPU architectures.

Platform	P_{CPU} Xeon E5-2670@2.60GHz	P_{GPU} Kepler K20m
$e^{op}_{T_u,P_i,L_a}$ [pJ]	327	54
$e^{byte}_{T_u,P_i,L_a}$ [pJ]	1700	324

The energy usage of the seven-point stencil shown in Algorithm 1 can be calculated based on both parameters depicted in Section 5.4 and the approximated

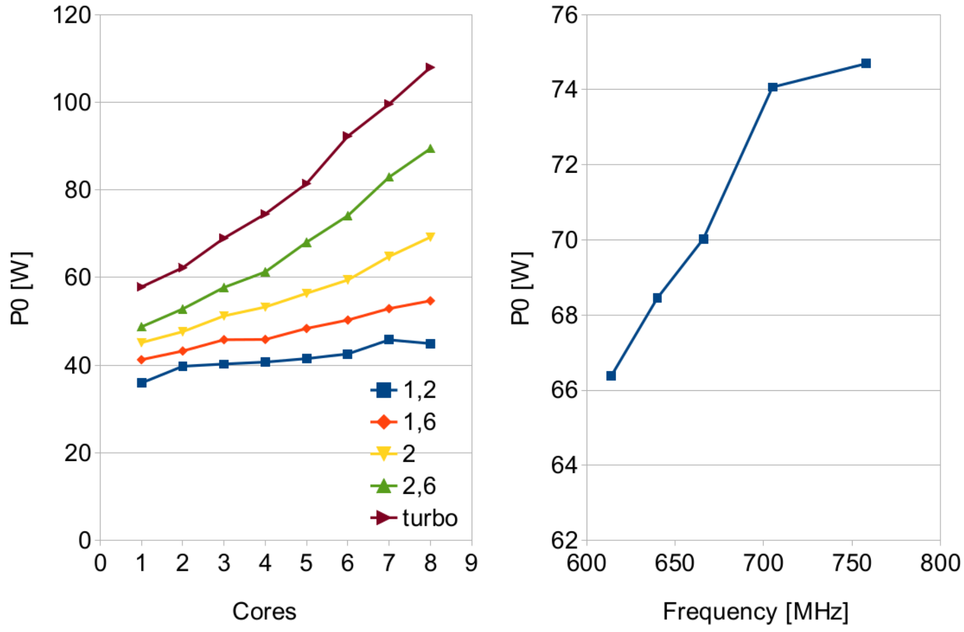


Figure 5.10: Constant power P0: left - the CPU, right - the GPU

energy coefficients, see the equations below:

$$e^{e,WC}_{T_u,P_{CPU},L_a} = 327\frac{pJ}{FLOP}*112MFLOP+1700\frac{pJ}{Byte}*1152MB+89W*37.5ms = 5.33J \tag{5.18}$$

$$e^{e,BC}_{T_u,P_{CPU},L_a} = 327\frac{pJ}{FLOP}*112MFLOP+1700\frac{pJ}{Byte}*256MB+89W*8.33ms = 1.21J \tag{5.19}$$

$$e^{e,WC}_{T_u,P_{GPU},L_a} = 54\frac{pJ}{FLOP}*112MFLOP+324\frac{pJ}{Byte}*1024MB+74W*5.78ms = 0.766J \tag{5.20}$$

$$e^{e,BC}_{T_u,P_{GPU},L_a} = 54\frac{pJ}{FLOP}*112MFLOP+324\frac{pJ}{Byte}*256MB+74W*1.48ms = 0.199J \tag{5.21}$$

The equations show the energy usage of the GPU is 6x to 7x lower than for the CPU.

Chapter 6
Solution methods

In this chapter, we introduce an exact method based on ILP to obtain the optimal solution for the formulated problem. We also present the heuristics with two objectives: to minimise the energy usage and load balance of the tasks to meet the deadline. We describe various computational experiments and simulation results for the ILP model on relatively small problem instances.

6.1 Exact method

Let us present a method based on ILP to obtain optimal solutions to the energy minimisation problem. In particular, this method is developed to have a reference for more advanced heuristics described in Section 6.2. Our method was inspired by the model proposed in [49]. The idea is to decompose the scheduling problem to two parallel subproblems. At first, the tasks are mapped to processors to minimise the maximum number of grid cells placed on each processor. Secondly, the number of communication rounds is minimised by employing an edge colouring model. The communication is executed in parallel between different pairs of processors in stencil computations. However, each processor can initiate a single communication link with another processor at a time. As a result, we have to employ several communication rounds to exchange all data. The number of communication rounds directly influences the communication time t^e, as each round costs some time. The reason for selecting the ILP solution is that the edge colouring problem is NP-hard [46] and [37]. For the task scheduling model, the set of edges is mapped to processors, where each edge (T_u, T_v) may be mapped to a single processor P_i or placed between two different processors P_i and P_j. In the first case, both endpoints T_u and T_v are mapped to P_i. In the second case, T_u is mapped to P_i and T_v to P_j or T_u is mapped to P_j and T_v to P_i. For each edge the *slots* $(P_i, P_j) \in E^P$ are provided and it is required that each edge be assigned to exactly one slot. If edge (T_u, T_v) is assigned to slot (P_i, P_j) then it starts in P_i and ends in P_j. If $P_i = P_j$ then (T_u, T_v) lies completely on P_i and is intra-processor, in all other cases it is inter-processor. The edge colouring model is used to minimise the number of communication rounds. If

the graph $G = (V^T, E^T)$ of tasks is mapped to the complete graph $K = (V^P, E^P)$ of m processors to form a new multigraph M, then each edge in M receives at least as many colours as its multiplicity demands and incident edges do not receive the same colour. Moreover, an edge can only receive a colour that is used.

Variables. For the integer programming model we introduce the following variables:

- (*Edge to slot* $x_{(T_u,T_v),(P_i,P_j)}$) The binary variable for all $(T_u, T_v) \in E^T$ and $(P_i, P_j) \in E^P$ equals 1 if and only if edge (T_u, T_v) is mapped to slot (P_i, P_j), and 0 otherwise
- (*Edge to colour* $y_{(P_i,P_j),c}$) For all $(P_i, P_j) \in K$ and $c \in C$, where $C = \{0, ..., \Delta(G)+\mu(G) - 1\}$, the binary variable equals 1 if edge (P_i, P_j) receives colour c in M, and 0 otherwise. The simplest choice for the number of potential colours to a colour multigraph is $|E|$. However, we can choose a smaller set based on [92, 102] that for any multigraph $G = (V, E)$ the chromatic index is $\chi'(G) \leq \Delta(G) + \mu(G)$
- (*Colour is used* z_c) For all $c \in C$ the binary variable equals to 1 if a colour c is used in the edge colouring of M and 0 otherwise
- (*Number of grid cells* g_{P_i}) This integer variable depicts for each processor P_i the number of allocated grid cells
- (*Processor idle time* $t^{idle}_{P_i}$) This variable for each processor P_i with the state L_a represents the idle time
- (*Total execution time* t^s) This variable indicates how much time it takes to finish the whole workload
- (*Energy used for communication* e^c) This variable represents the total energy used for the inter-processor communication.

Constraints. The model employs several types of constraints:

- (*Map edge to single slot*) Each edge $(T_u, T_v) \in E^T$ must be mapped to exactly one slot

$$\sum_{(P_i,P_j)\in E^P} x_{(T_u,T_v),(P_i,P_j)} = 1 \tag{6.1}$$

- (*Restrict slots*) Mapping edge (T_u, T_v) to slot (P_i, P_j) restricts the slots to which edges in $\delta((T_u, T_v))$ can be mapped. Edges in $\delta^+(T_u)$ must start in P_i and edges in $\delta^-(T_u)$ must end there. Likewise, edges in $\delta^+(T_v)$ must start in P_j and edges in $\delta^-(T_v)$ must end there:

$$\sum_{k\in\mathcal{P}} x_{(T_u,T_v),(P_i,P_j)} - \sum_{k\in\mathcal{P}} x_{f,(P_i,P_j)} = 0 \tag{6.2}$$

$$\sum_{k\in\mathcal{P}} x_{(T_u,T_v),(P_i,P_j)} - \sum_{k\in\mathcal{P}} x_{f,(P_j,P_i)} = 0 \tag{6.3}$$

$$\sum_{k\in\mathcal{P}} x_{(T_u,T_v),(P_j,P_i)} - \sum_{k\in\mathcal{P}} x_{f,(P_i,P_j)} = 0 \tag{6.4}$$

$$\sum_{k\in\mathcal{P}} x_{(T_u,T_v),(P_j,P_i)} - \sum_{k\in\mathcal{P}} x_{f,(P_j,P_i)} = 0 \tag{6.5}$$

These constraints are for all $P_i \in V^P$ and $(T_u, T_v) \in E^T$, where $f \in \delta^+(T_u)$ is for (6.2), $f \in \delta^-(T_u)$ is for (6.3), $f \in \delta^+(T_v)$ is for (6.4), $f \in \delta^-(T_v)$ is for (6.5),

- (*Control the number of grid cells*) This constraint controls the number of the grid cells allocated for each $P_i \in V^P$. The sum of grid cells mapped to processor P_i is given by

$$\sum_{(T_u,T_v)\in E^T} \sum_{j\in V^P} (d_{T_u}/deg_{T_u} * x_{(T_u,T_v),(P_i,P_j)} + d_{T_v}/deg_{T_v} * x_{(T_u,T_v),(P_j,P_i)}) \leq c_{P_i} \tag{6.6}$$

  for all $P_i \in V^P$

- (*Number of colours not less than multiplicity*) This constraint requires that each edge in M receive at least as many colours as its multiplicity demands. Each edge models time required to exchange single grid cell between processors P_i and P_j:

$$\sum_{(T_u,T_v)\in E^T} (x_{(T_u,T_v),(P_i,P_j)} * t^c_{(P_i,P_j)} * C_{(T_u,T_v)} + x_{(T_u,T_v),(P_j,P_i)} * t^c_{(P_j,P_i)} * C_{(T_u,T_v)}) \leq \underset{c\in C}{y_{(P_i,P_j),c}} \tag{6.7}$$

- (*Incident edges receive different colours*) This requires that incident edges not receive the same colour in the edge colouring of M and that an edge can only receive a colour that is used:

$$\sum_{P_j \neq P_i, P_J \in V^P} y_{(P_i,P_j),c} \leq z_c \tag{6.8}$$

- (*Restrict memory capacity for each processor*) This constraint restricts the number of grid cells allocated for each processor P_i:

$$g_{P_i} \leq r_{P_i} \tag{6.9}$$

- (*Control energy used for communication*) The sum of energy used for the inter-processor communication is depicted as

$$\sum_{(T_u,T_v)\in E^T} \sum_{P_i \neq P_j, (P_i,P_j)\in E^P} x_{(T_u,T_v),(P_i,P_j)} * C_{(T_u,T_v)} * e^c_{(P_i,P_j)} \leq e^c \tag{6.10}$$

- (*Control execution time*) These two constraints calculate the total execution time t^s using the maximum value from the computation and the communica-

tion time. As described in Section 5.2 the computation and the communication are done in parallel:

$$t^{e}_{T_u,P_i,L_a} * c_{P_i} \leq t^{s} \quad (6.11)$$

$$\sum_{c \in C} z_c \leq t^{s} \quad (6.12)$$

- (*Control processor's idle time*) This constraint controls the idle time for all $P_i \in V^P$:

$$t^{s} - t^{c}_{T_u,P_i,L_a} * c_{P_i} \leq t^{idle}_{P_i} \quad (6.13)$$

- (*Deadline*) This inequality restricts the execution time:

$$t^{s} \leq t^{d} \quad (6.14)$$

Table 6.1: Number of variables and constraints that formulate the ILP problem.

	ILP
x variables	$\|E \times \mathcal{P} \times \mathcal{P}\|$
y variables	$\|C \times \mathcal{P} \times \mathcal{P}\|$
z variables	$\|C\|$
x constraints	$\|E\|$
y constraints	$\|\mathcal{P} \times \mathcal{P}\|$
z constraints	$\|\mathcal{P}\|$

Table 6.1 shows the number of variables and constraints that formulate the ILP model.

Optimisation objective. Finally, the objective of the model is to minimise the energy cost:

$$\sum_{P_i \in V^P} (e^{e}_{T_u,P_i,L_a} * g_{P_i} + t^{idle}_{P_i} * P^{idle}_{P_i}) + e^{c} \quad (6.15)$$

6.2 Heuristic algorithms

In this section, we introduce new heuristics taking into account the relevance of energy efficiency issues in the next generation of the high-end supercomputers. In our approach we consider energy-aware stencil workload scheduling on heterogeneous architectures with two following objectives:

- minimise the energy usage;
- load balance of the tasks to meet the deadline.

6.2.1 Load Balancing

First strategy focuses on balancing the load between processors and does not consider the communication dependencies. These heuristics are usually relatively fast and straightforward as they act online on the workload.

The algorithm distributes tasks to processors while keeping the maximal load small and not exceeding the deadline. This strategy is called Load Balancing, see Algorithm 2. We start with processor p_0 and assign tasks to this processor until its size is at least $w_V * r_i / \sum_{p \in P} r_p$. Then we move to the next processor and repeat the procedure. The limit $w_V * r_i / \sum_{p \in P} r_p$ stems from the fact that in perfect balancing of tasks there is one processor that has this many grid cells. This limit is a modification of a limit $w_V / |P|$ for a homogeneous processor, as we consider the speed r_p of each processor. The time complexity of the algorithm is $\mathrm{O}\big(|V|\big)$ to assign all tasks to processors. The algorithm is sensitive to the order in which the tasks and processors are selected.

Algorithm 2 Load Balancing

```
1: procedure LoadBalancing
   Input: A set 𝒯 of tasks and the processor set 𝒫.
   Output: A mapping m.
2:     for i = 0, ..., |𝒫| − 1 do
3:         Set s = 0
4:         while s ≤ w_V * r_i / Σ_{p∈𝒫} r_p do
5:             Remove the first task u from 𝒯
6:             Set m(u) = p_i and s = s + w_u
7:         end while
8:     end for
9: end procedure
```

6.2.2 Degree Minimisation

Algorithms described in this section attempt to include communication overhead in the scheduling process. They try to find such a schedule that the resulting multi-graph yields a small chromatic index. Since finding that the chromatic index is an NP-complete problem [46], the algorithms employ different approximation methods to minimise it.

In this algorithm task, u with the lowest number of unmapped edges is assigned to the current processor p. Based on the equations $\chi'(G) \leq \Delta(G) + \mu(G)$ and $\chi'(G) \leq \lfloor 3 * \Delta(G)/2 \rfloor$ the chromatic index $\chi'(G)$ of any multigraph G depends on the max degree. Thus, when task u is assigned to processor p, then each incident edge to this task not mapped to p increases the current degree of p by one. The

neighbours of the task u that are mapped to another processor $k \neq p$ also increase the degree of p, but they are not considered in this algorithm. Therefore, the array $deg(u)$ is used to keep the number of unmapped edges for each task u. It is the number by which the degree of processor p would increase if the task u was mapped to it. If two tasks have the same number of unmapped edges, then the task with the smallest computational load is selected. In other words, it is the number of additional grid points by which computational load on the processor p would exceed the perfect load $w_V * r_i / \sum_{p \in P} r_p$ if the task u was mapped to p. The running time of Algorithm 3 is $O\Big(|V^2|\Big)$. The time needed to find the task with the smallest computational load takes $O\Big(|V|\Big)$, whereas the while loop is executed $O\Big(|V|\Big)$ times.

Algorithm 3 Degree Minimisation

```
1:  procedure DEGREE MINIMISATION
    Input: A set 𝒯 of tasks and the processor set 𝒫.
    Output: A mapping m.
2:      for u ∈ 𝒯 do
3:          deg(u) = deg_G(u)
4:      end for
5:      for i = 0, ..., |𝒫| − 1 do
6:          Set s = 0
7:          while s ≤ w_V * r_i / Σ_{p∈𝒫} r_p do
8:              Find u' = argmin{deg(u) : u ∈ 𝒯}.
9:                                  ▷ If there are multiple tasks that attain this
10:                     ▷ minimum pick the one with smallest computational load.
11:             Remove task u' from set 𝒯
12:             Set m(u') = p_i and s = s + w_{u'}
13:             for v ∈ N(u') ∩ 𝒯 do
14:                 deg(v) = deg(v) − μ_{u'v}
15:             end for
16:         end while
17:     end for
18: end procedure
```

6.2.3 Multicut Minimisation

This algorithm estimates the chromatic index for a multigraph based on the complete number of edges $|E|$. The previous Algorithm 3 is modified to obtain a minimal multicut, see Algorithm 4. The task u with the smallest number of unscheduled neighbours is found to be mapped on the current processor p. In line three the $deg(u)$ is initialised with the number of unscheduled neighbours. Each scheduled task u' decreases the $deg(v)$ for each unscheduled neighbour v of u'. The time complexity of the algorithm is equal to $O\Big(|V^2|\Big)$.

Algorithm 4 Multicut Minimisation

```
1: procedure MULTICUT MINIMISATION
   Input: A set 𝒯 of tasks and the processor set 𝒫.
   Output: A mapping m.
2:     for u ∈ 𝒯 do
3:         deg(u) = |N(u) ∩ 𝒯|
4:     end for
5:     for i = 0, ..., |𝒫| − 1 do
6:         Set s = 0
7:         while s ≤ w_V * r_i / Σ_{p∈P} r_p do
8:             Find u' = argmin{deg(u) : u ∈ 𝒯}.
9:             Remove task u' from set 𝒯
10:            Set m(u') = p_i and s = s + w_{u'}
11:            for v ∈ N(u') ∩ 𝒯 do
12:                deg(v) = deg(v) − 1
13:            end for
14:        end while
15:    end for
16: end procedure
```

6.2.4 Neighbours Accumulation

In this algorithm the unmapped task u with the highest number of neighbours on the currently selected processor p is chosen, see Algorithm 5. This policy tries to yield most of the communication edges of the grid graph intra-processor. The array N records the number of neighbours which the task u has on the processor p. In line 10 the task with the most neighbours on the processor p is selected. However, at the end of the inner while loop (line 12), when the processor p is almost full, a different strategy is employed. The task u connected to the subgraph mapped to p with a minimum number of neighbours not on p is selected. The load factor $f \in [0,1]$ is introduced to recognise when the processor is almost full. While $s \leq f * w_V * r_i / \sum_{p \in \mathcal{P}} r_p$ the tasks with the maximum number of neighbours on the current processor p are selected, whereas $s \geq f * w_V * r_i / \sum_{p \in \mathcal{P}} r_p$ the tasks with the minimum number of neighbours not on p are picked. When no unmapped task is adjacent to the tasks currently mapped to p, the task with the maximum degree is preferred. Additionally, for the strategy defined in line 12, the task with the minimum degree among the unmapped ones is selected. To find the task in lines 10 and 12 takes $O(|V|)$ time. The while loops are executed $|V|$ times, thus the whole algorithm runs in time $O(|V^2|)$.

Algorithm 5 Neighbours Accumulation

```
1: procedure Neighbours Accumulation
   Input: A set T of tasks and the processor set P.
   Output: A mapping m.
2:   Set i = 0
3:   while T ≠ ∅ and i ≤ |P| do
4:     for u ∈ T do
5:       N(u) = 0
6:     end for
7:     Set s = 0
8:     while s ≤ w_V * r_i / Σ_{p∈P} r_p and T ≠ ∅ do
9:       if s ≤ f * w_V * r_i / Σ_{p∈P} r_p then
10:        Find u' = argmax{N(u) : u ∈ T}.
11:      else
12:        Find u' = argmin{deg_G(u) − N(u) : u ∈ T, N(u)}.
13:      end if
14:                              ▷ If there are multiple tasks that attain this
15:                  ▷ minimum pick the one with smallest computational load.
16:      Remove task u' from set T
17:      Set m(u') = p_i and s = s + w_{u'}
18:      For each v ∈ T that is adjacent to u' set N(v) = N(v) + 1
19:    end while
20:  end while
21: end procedure
```

6.3 Computational experiments

6.3.1 Simulation setup

For the purpose of computational experiments a new simulator has been designed and implemented to validate our models and calculate the total execution time, energy usage and the number of communication rounds (colours). The simulator is initialised with the following data:

1. a text file with a workload dependency graph
2. a text file with processor topology
3. the type of scheduling strategy used: ILP or heuristic.

The simulation instances include two different real-world simulation grids. These grids are related to the weather simulations problems. The connection topology of points on each grid is defined by a 3D seven-point stencil depicted in Figure 3.1.

Table 6.2 outlines the properties of the test instances. The first grid called Cuboid (Figure 6.1) was used to simulate decaying turbulence of a homogeneous

incompressible fluid, whereas the second grid called Sphere (Figure 6.2) was used as a benchmark for the atmospheric circulation models. The connections in the horizontal direction for the Sphere grid are periodical. Figure 6.3 shows the example of the decomposed Cuboid grid with the connection dependencies. Each number represents the block identifier that is later mapped to a specific processor. The grids are mapped to a single node with three different configurations of the processors: CPU-CPU, CPU-GPU and 2xCPU-2xGPU to analyse the quality of the ILP model and the heuristics. The simulated CPU is Intel Xeon E5-2670 Sandy Bridge eight core processor, and the GPU is Nvidia Kepler K20m. Figure 6.4 presents the node topology with four processors. The CPU and GPU frequencies are set to default values in all algorithms. The GPU operates at 705MHz of the core clock and 2600MHz of the memory clock, where the CPU operates at 2.6GHz of the core clock. The parameters used in all test runs are shown in Table 6.3. The values of the parameters are obtained based on the methodology described in Sections 5.1 and 5.3.

Table 6.2: Properties of the simulated grids.

Name	#Blocks	#Edges	Block size	Grid size
Cuboid	128	608	262144	512x256x256
Sphere	128	704	262144	512x256x256

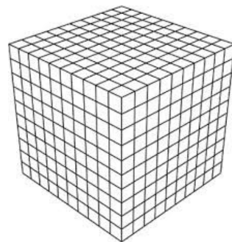


Figure 6.1: Cuboid

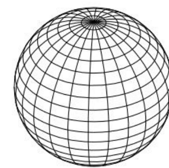


Figure 6.2: Sphere

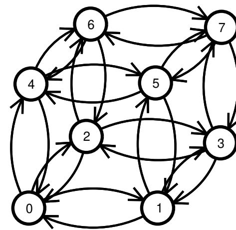


Figure 6.3: Graph of stencil tasks with the connection dependencies

6.3.2 Simulation results

First, we show the results for the ILP model, see Tables 6.4 and 6.5, where the first column presents the configurations used. Each configuration is simulated with different deadlines. The deadline is provided as an input parameter. We reduce its value

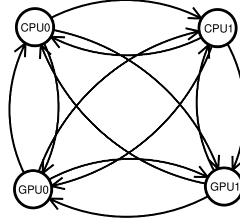


Figure 6.4: Graph of processors

Table 6.3: Parameters setup.

Parameter	CPU	GPU
$t^e_{p,k}$	$1 * 10^{-9} s$	$1 * 10^{-9} s$
$e^e_{p,k}$	$1.36 * 10^{-7} J$	$1.36 * 10^{-7} J$
$t^c_{u,p,l}$	$8.33 * 10^{-10} s$	$1.06 * 10^{-10} s$
$e^c_{u,p,l}$	$2.9 * 10^{-8} J$	$5.5 * 10^{-9} J$
P^{idle}_p	10W	30W
$P0_{u,p,l}$	90W	74W

to the point where the ILP model cannot generate a feasible solution. The subsequent columns provide the number of colours used in a multigraph, the total energy consumed, the energy used for computations, the energy used for communication, and the time elapsed. The number of colours used in the graph colouring provides information about the number of communication rounds employed. As the results show, the decreasing deadlines improve the computation times; however, they increase the energy usage. This is especially true for the heterogeneous configurations where the energy usage grows up to 7x from the extended deadline to the shortest one. A shorter deadline forces usage of the next processing unit and, as a result, it requires more energy to communicate. For example, for the *2xCPU-2xGPU* configuration 88% of energy is consumed by the communication. For this reason, it is crucial to efficiently distribute the stencil tasks to reduce the number of communication rounds between the processing units. However, as we can see, it is beneficial to use the heterogeneous configurations; as we switch from the *CPU-CPU* configuration to the *2xCPU-2xGPU* configuration, both the computation time and energy costs decrease by 87% and 57%, respectively, for the Cuboid grid. Similarly, the computation time for the Sphere grid decreases by 87%, whereas the energy usage decreases by 42%. Higher energy usage for the Sphere grid is caused by the periodic connections of tasks on the I and J boundaries. For single-node configurations we can notice that the maximum computation time $t^c_{u,p,l}$ among the processing units is a bottleneck for the total execution time t^t. Still, we can expect that the limiting factor will be the communication time for the multi-node configurations.

The quality of all the four heuristics described in Section 6.2 is presented. The obtained results are presented in Tables 6.6, 6.7, 6.8, 6.9, 6.10, and 6.11, where Algorithm 1 is tested with four different sorting orders of the tasks: random (RD),

Table 6.4: ILP on Cuboid with all configurations.

Arch.	t^d[ms]	#Col.	E[J]	e^c[%]	e^e[%]	t^t[ms]
CPU-CPU	27,90	0	3,76	100	0	27,90
	26,97	20	4,86	77	23	26,81
	14,00	32	5,27	66	34	13,95
CPU-GPU	3,58	0	0,48	100	0	3,58
	3,46	20	1,70	35	65	3,44
	3,34	28	2,23	30	70	3,33
	3,22	34	2,65	29	71	3,22
2xCPU-2xGPU	3,58	0	0,62	100	0	3,58
	3,46	20	1,73	36	64	3,44
	3,34	28	2,16	28	72	3,33
	3,22	32	2,26	21	79	1,79

Table 6.5: ILP on Sphere with all configurations.

Arch.	t^d[ms]	#Col.	E[J]	e^c[%]	e^e[%]	t^t[ms]
CPU-CPU	26,97	30	5,41	69	31	26,81
	25,92	40	5,94	63	37	25,28
	25,02	48	6,37	58	42	24,41
	24,13	56	6,78	54	46	22,67
	14,00	64	7,05	49	51	13,95
CPU-GPU	3,60	0	0,48	100	0	3,58
	3,46	30	2,26	26	74	3,44
	3,35	38	2,79	24	76	3,33
	3,23	46	3,32	23	77	3,22
2xCPU-2xGPU	3,58	0	0,62	100	0	3,58
	3,46	30	2,28	27	73	3,44
	3,34	38	2,72	22	78	3,33
	3,22	46	3,16	19	81	3,22
	3,11	54	3,60	16	84	3,11
	2,99	56	3,69	16	84	2,91
	2,66	64	4,05	12	88	1,79

Table 6.6: Heuristics on Cuboid with the CPU-CPU configuration.

Algorithm	#Edg.	#Col.	E[J]/gap	t^t[ms]/gap
Alg_1-RD	1224	306	20,53/289,49%	13,95/0,00%
Alg_1-IJK	256	64	**7,05/33,81%**	13,95/0,00%
Alg_1-JIK	256	64	**7,05/33,81%**	13,95/0,00%
Alg_1-KIJ	256	64	**7,05/33,81%**	13,95/0,00%
Alg_2	256	64	**7,05/33,81%**	13,95/0,00%
Alg_3	256	64	**7,05/33,81%**	13,95/0,00%
Alg_4-0,1	256	64	**7,05/33,81%**	13,95/0,00%
Alg_4-0,2	256	64	**7,05/33,81%**	13,95/0,00%
Alg_4-0,3	256	64	**7,05/33,81%**	13,95/0,00%
Alg_4-0,4	256	64	**7,05/33,81%**	13,95/0,00%
Alg_4-0,5	256	64	**7,05/33,81%**	13,95/0,00%
Alg_4-0,6	256	64	**7,05/33,81%**	13,95/0,00%
Alg_4-0,7	256	64	**7,05/33,81%**	13,95/0,00%
Alg_4-0,8	256	64	**7,05/33,81%**	13,95/0,00%
Alg_4-0,9	256	64	**7,05/33,81%**	13,95/0,00%
Alg_4-1.0	256	64	**7,05/33,81%**	13,95/0,00%

Table 6.7: Heuristics on Cuboid with the CPU-GPU configuration.

Algorithm	#Edg.	#Col.	E[J]/gap	t^t[ms]/gap
Alg_1-RD	504	126	7,85/195,85%	3,48/8,32%
Alg_1-IJK	192	48	3,51/32,29%	3,48/8,32%
Alg_1-JIK	160	40	**3,06/15,52%**	3,48/8,32%
Alg_1-KIJ	160	40	**3,06/15,52%**	3,48/8,32%
Alg_2	192	48	3,51/32,29%	3,48/8,32%
Alg_3	192	48	3,51/32,29%	3,48/8,32%
Alg_4-0,1	224	56	3,96/49,07%	3,48/8,32%
Alg_4-0,2	200	50	3,62/36,49%	3,48/8,32%
Alg_4-0,3	200	50	3,62/36,49%	3,48/8,32%
Alg_4-0,4	200	50	3,62/36,49%	3,48/8,32%
Alg_4-0,5	200	50	3,62/36,49%	3,48/8,32%
Alg_4-0,6	200	50	3,62/36,49%	3,48/8,32%
Alg_4-0,7	192	48	3,51/32,29%	3,48/8,32%
Alg_4-0,8	192	48	3,51/32,29%	3,48/8,32%
Alg_4-0,9	176	44	3,29/23,90%	3,48/8,32%
Alg_4-1.0	160	40	**3,06/15,52%**	3,48/8,32%

Table 6.8: Heuristics on Cuboid with the 2xCPU-2xGPU configuration.

Algorithm	#Edg.	#Col.	E[J]/gap	t^t[ms]/gap
Alg_1-RD	520	318	22,00/870,23%	1,74/-2,68%
Alg_1-IJK	576	128	8,86/290,69%	1,74/-2,68%
Alg_1-JIK	480	112	7,52/231,75%	1,74/-2,68%
Alg_1-KIJ	480	112	7,52/231,75%	1,74/-2,68%
Alg_2	576	128	8,86/290,69%	1,74/-2,68%
Alg_3	576	128	8,86/290,69%	1,74/-2,68%
Alg_4-0,1	568	116	8,75/285,77%	1,74/-2,68%
Alg_4-0,2	504	92	7,85/246,48%	1,74/-2,68%
Alg_4-0,3	584	124	8,97/295,60%	1,74/-2,68%
Alg_4-0,4	480	100	7,52/231,75%	1,74/-2,68%
Alg_4-0,5	480	100	7,52/231,75%	1,74/-2,68%
Alg_4-0,6	480	100	7,52/231,75%	1,74/-2,68%
Alg_4-0,7	472	98	7,41/226,84%	1,74/-2,68%
Alg_4-0,8	456	92	7,19/217,01%	1,74/-2,68%
Alg_4-0,9	432	84	**6,85/202,28%**	1,74/-2,68%
Alg_4-1.0	432	84	**6,85/202,28%**	1,74/-2,68%

Table 6.9: Heuristics on Sphere with the CPU-CPU configuration.

Algorithm	#Edg.	#Col.	E[J]/gap	t^t[ms]/gap
Alg_1-RD	1408	352	23,09/227,39%	13,95/0,00%
Alg_1-IJK	256	64	**7,05/0,00%**	13,95/0,00%
Alg_1-JIK	256	64	**7,05/0,00%**	13,95/0,00%
Alg_1-KIJ	512	128	10,62/50,53%	13,95/0,00%
Alg_2	256	64	**7,05/0,00%**	13,95/0,00%
Alg_3	256	64	**7,05/0,00%**	13,95/0,00%
Alg_4-0,1	256	64	**7,05/0,00%**	13,95/0,00%
Alg_4-0,2	256	64	**7,05/0,00%**	13,95/0,00%
Alg_4-0,3	256	64	**7,05/0,00%**	13,95/0,00%
Alg_4-0,4	256	64	**7,05/0,00%**	13,95/0,00%
Alg_4-0,5	256	64	**7,05/0,00%**	13,95/0,00%
Alg_4-0,6	256	64	**7,05/0,00%**	13,95/0,00%
Alg_4-0,7	256	64	**7,05/0,00%**	13,95/0,00%
Alg_4-0,8	256	64	**7,05/0,00%**	13,95/0,00%
Alg_4-0,9	256	64	**7,05/0,00%**	13,95/0,00%
Alg_4-1.0	256	64	**7,05/0,00%**	13,95/0,00%

Table 6.10: Heuristics on Sphere with the CPU-GPU configuration.

Algorithm	#Edg.	#Col.	E[J]/gap	t^t[ms]/gap
Alg_1-RD	576	144	9,53/186,63%	3,48/8,32%
Alg_1-IJK	256	64	4,40/32,50%	3,48/8,32%
Alg_1-JIK	192	48	**3,51/5,70%**	3,48/8,32%
Alg_1-KIJ	320	80	5,29/59,31%	3,48/8,32%
Alg_2	256	64	4,40/32,50%	3,48/8,32%
Alg_3	256	64	4,40/32,50%	3,48/8,32%
Alg_4-0,1	296	74	4,96/49,25%	3,48/8,32%
Alg_4-0,2	304	76	5,07/52,61%	3,48/8,32%
Alg_4-0,3	304	76	5,07/52,61%	3,48/8,32%
Alg_4-0,4	312	78	5,18/55,96%	3,48/8,32%
Alg_4-0,5	320	80	5,29/59,31%	3,48/8,32%
Alg_4-0,6	328	82	5,40/62,66%	3,48/8,32%
Alg_4-0,7	336	84	5,51/66,01%	3,48/8,32%
Alg_4-0,8	336	84	5,51/66,01%	3,48/8,32%
Alg_4-0,9	320	80	5,29/59,31%	3,48/8,32%
Alg_4-1.0	320	80	5,29/59,31%	3,48/8,32%

Table 6.11: Heuristics on Sphere with the 2xCPU-2xGPU configuration.

Algorithm	#Edg.	#Col.	E[J]/gap	t^t[ms]/gap
Alg_1-RD	1664	350	24,01/492,79%	1,74/-2,68%
Alg_1-IJK	704	160	10,64/162,77%	1,74/-2,68%
Alg_1-JIK	704	160	10,64/162,77%	1,74/-2,68%
Alg_1-KIJ	800	760	11,98/195,77%	1,74/-2,68%
Alg_2	704	160	10,64/162,77%	1,74/-2,68%
Alg_3	704	160	10,64/162,77%	1,74/-2,68%
Alg_4-0.1	744	164	11,20/176,52%	1,74/-2,68%
Alg_4-0,2	728	158	10,97/171,02%	1,74/-2,68%
Alg_4-0,3	712	152	10,75/165,52%	1,74/-2,68%
Alg_4-0,4	712	156	10,75/165,52%	1,74/-2,68%
Alg_4-0,5	720	158	10,86/168,27%	1,74/-2,68%
Alg_4-0,6	720	158	10,86/168,27%	1,74/-2,68%
Alg_4-0,7	704	158	10,64/162,77%	1,74/-2,68%
Alg_4-0,8	704	158	10,64/162,77%	1,74/-2,68%
Alg_4-0,9	672	144	**10,19/151,77%**	1,74/-2,68%
Alg_4-1,0	672	144	**10,19/151,77%**	1,74/-2,68%

IJK indices, JIK indices and KIJ indices. The grid indices can order the tasks depending on their location within the grid. This order may influence the number of edges mapped between different processors.

For Algorithm 4, the first column in Tables contains the value of the load factor f, which depicts when the processor is almost full. This algorithm is tested with different values of this parameter. The second to last columns show the number of edges in the returned scheduling, the number of colours used in the obtained multigraph and the objective values for the energy and time. The gap is defined as a difference between the optimal solution o' and the solution o^* returned by the algorithm:

$$gap(o^*, o') = (o^* - o')/o' \tag{6.16}$$

The optimal solution with the shortest deadline is selected as a base for the comparison. In other words, the results are compared to the feasible solution with the lowest possible computational time and minimal energy obtained by the ILP model. Tables from 6.6 to 6.11 show that the time t^t is all the same. All heuristics are based on the idea of load balancing, where the computations of the tasks are well balanced between processors. For each grid configuration the final schedule obtains the same computation time. The communication time between heuristics is different, as the obtained schedules provide a different number of communication rounds. The communication time is shorter than the computation time, and both are done in parallel. As a result, the communication time does not influence the time t^t. However, the number of communication rounds strongly influences energy consumption. Tables 6.6 and 6.9 show that almost all heuristics except for *Alg_1-RD* are able to schedule stencil tasks with a near-optimal solution for homogeneous hardware configurations with two processors. What is more, the results show that the heuristics that target the balanced load provide good solutions for simple configurations with two processors. The Load Balancing algorithm *Alg_1* produces an efficient distribution depending on the sorting order of the input tasks. The order based on JIK indices minimises the number of communication rounds for both grids with two processors. It is beneficial to use the heuristics that take into account the communication penalty with four processors. The algorithm *Alg_4* provides good schedules for the four processors as it tries to yield most of the communication edges of the task graph intra-processor. The quality of the schedule depends on the load factor which determines when to switch the mapping from the task with the most neighbours on the current processor to the task with a minimum number of neighbours, not on the current processor. Let us take as an example the $5x4x5$ grid with 100 blocks distributed on a node with a single CPU and two GPUs where the 3D Laplacian stencil is employed. Figure 6.5a shows the schedule from *Alg_1* with the best performing JIK order. The blocks are distributed horizontally according to the JIK order. Figure 6.5b shows the output from *Alg_4* with the load factor equal to 0.9. The blocks scheduled to the CPU are distributed vertically within the computational grid, whereas those scheduled to GPUs are distributed horizontally. *Alg_4* and the

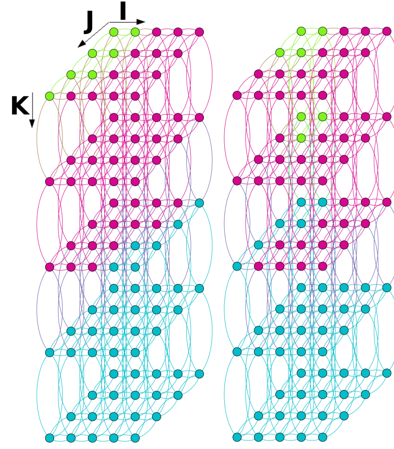


Figure 6.5: Comparison of schedule between *Alg_1* and *Alg_4*. The colours represent the scheduling of blocks to the processors: red - GPU00, blue - GPU01 and green - CPU00. Left - output from *Alg_1*, right - output from *Alg_4*.

rest of the algorithms (*Alg_2* and *Alg_3*) are able to mix the spatial distribution of the blocks. The energy cost is $4.65J$ and $4.43J$ for *Alg_1* and *Alg_4*, respectively. 5% of the energy is saved by reducing the number of communication rounds.

The presented heuristics may be applied to the distribution of stencil computations between the processing units defined on the Cartesian grids. These grids may be 2D or 3D with or without periodic boundaries.

Table 6.12: The average execution time (us) of the investigated ILP model and heuristics for the 2xCPU-2xGPU configuration.

	ILP	Alg1	Alg2	Alg3	Alg4
Cuboid	3118×10^6	32	78	52	55
Sphere	$281\,836 \times 10^6$	33	98	57	65

Table 6.12 shows the average execution time of the investigated ILP model and heuristics for the 2xCPU-2xGPU configuration with previously described grid setups. As one can notice, the time to find the optimal solutions is seven orders of magnitude larger than the time of heuristics.

6.3.3 Verification of energy model

This section contains the experimental comparison of the energy usage model used in the simulator with the real measurements. Figures 6.6 and 6.7 present the comparison of energy usage between the proposed model and the real measurements. The results are obtained for the Intel Xeon processor and the Nvidia K20m accelerator, respectively, for the seven-point stencil defined on the grid with 256^3 points. Table 6.13 contains the energy usage for all examined heuristics and the ILP model for

the 2xCPU-2xGPU configuration of processors defined on the Cube grid presented in Section 6.3.1. Table 6.14 summarizes their accuracy.

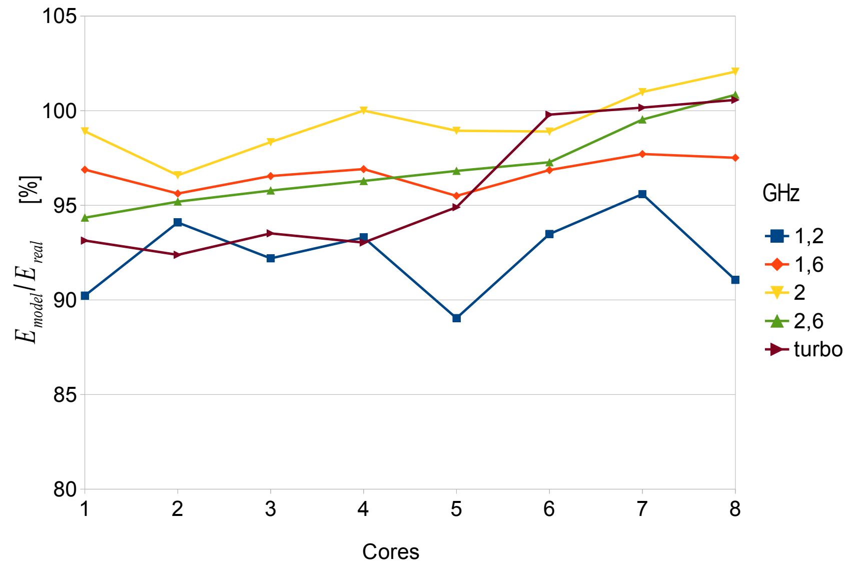


Figure 6.6: Comparison of accuracy (%) between the proposed model and the real measurements on the Intel Xeon E5-2670@2.6GHz processor.

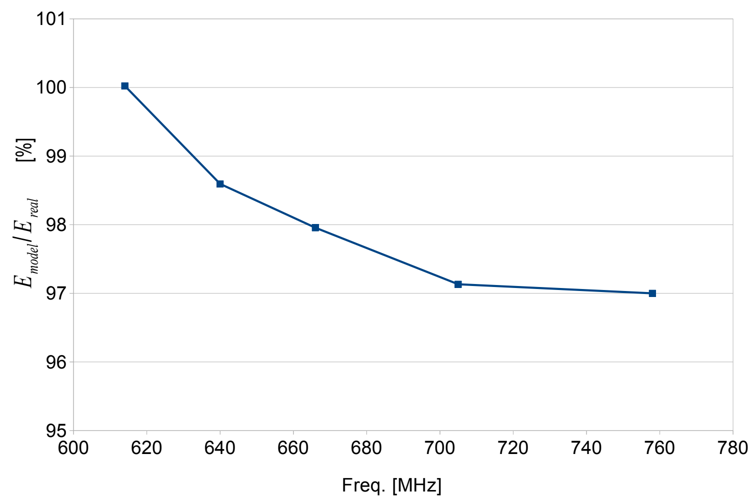


Figure 6.7: Comparison of accuracy (%) between the proposed model and the real measurements on the Nvidia K20m accelerator.

Table 6.13: Energy usage (J) of the proposed model and the real measurements for the investigated ILP model and heuristics.

	ILP	Alg1	Alg2	Alg3	Alg4
E_{real}	2,29	7,23	8,56	6,56	8,56
E_{model}	2,27	7,53	8,86	6,86	8,86

Table 6.14: Comparison of accuracy (%) between the proposed model and the real measurements for the investigated ILP model and heuristics.

	ILP	Alg1	Alg2	Alg3	Alg4
E_{model}/E_{real}	99	104	103	105	103

As can be observed, the accuracy of the presented model is high and visibly exceeds 90%. The results suggest that applying the time and energy models while verifying different scheduling policies does not deteriorate overall results. This concludes that the described environment can be used to simulate the heterogeneous computer system.

Chapter 7

Task Movement algorithm

The considered scheduling problem, as a generalisation of the NP-hard problem considered in [26], needs to be solved within an acceptable time frame using efficient heuristics. In previous section, we discussed results achieved by simple heuristics, namely Load Balancing (LB), Degree Minimisation (DM), Multicut Minimisation (MM) and Neighbours Accumulation (NA), to minimise the energy usage within a given deadline of parallel stencil computations. However, all the developed heuristics focused only on a single computing node with heterogeneous processors. In this section, however, which is built upon the results and conclusions from initial research experiments, further research is presented which has led to the development and experimental verification of a new algorithm Task Movement (TM) [27] to solve the stencil scheduling problem efficiently in more complicated and realistic multi-node HPC setups. Thus, to create the topology-aware TM algorithm, relevant additional parameters have been taken into account to deal with communication topology among heterogeneous processors linked together at the inter-node and intra-node levels. Finally, in this section, we present obtained simulation results by TM for large problem instances, crated out of high-end supercomputers real setups, and demonstrate its performance achievements.

7.1 Single-node setup and new requirements

To better understand our new TM algorithm, it is essential first to remind all the task and processor parameters in our model that have been used by the proposed heuristics optimised for single-node setups:

- $t^e_{P_i,L_a}$ the computation time for the single grid cell;
- $\mathcal{C} = \{c_1, c_2, \ldots, c_h\}$ the set of available cores;
- $\mathcal{F} = \{f_1, f_2, \ldots, f_g\}$ the set of available frequencies;
- $\mathcal{L} = \{(f, c) : f \in \mathcal{F} \wedge c \in \mathcal{C}\}$ the set of states, where $L_a \in \mathcal{L}$ is a selected state;

- b_{P_i,L_a} the sustained bandwidth to the main memory in bytes per second;
- $P^{perf}_{P_i,L_a}$ the performance in the floating-point operations per second;
- d_{T_u} the number of grid cells to process;
- W_{T_u,P_i} the number of arithmetic operations per grid cell;
- Q_{T_u,P_i} the number of required bytes to update a grid cell;
- $t^e_{T_u,P_i,L_a} = max(O_{T_u,P_i}/P_{P_i,L_a}; B_{T_u,P_i}/b_{P_i,L_a})$ where O_{T_u,P_i} is the number of arithmetic operations to update the stencil task T_u and B_{T_u,P_i} is the number of bytes to update the stencil task T_u;
- $deg(T_u)$ the degree of task T_u;
- $N(T_u)$ the number of neighbours of task T_u.

The above parameters characterise the task graph and processor speed. However, they do not describe the communication topology among heterogeneous processors linked together at the inter-node and intra-node levels. Therefore, during extensive research studies, we proposed a new heuristic algorithm taking into account different network and communication topologies to better predict both the runtime and the energy usage. A set of commonly used metaheuristic optimisation algorithms and techniques can be considered to produce good solutions in a reasonable time, namely: Tabu Search (TS) [39], Genetic Algorithms (GA) [93] and Simulated Annealing (SA) [54]. However, based on our initial computational studies, we have selected and focused on TS to solve the stencil scheduling problem efficiently in a new, multi-node setup.

7.2 Algorithm for multiple-node setup

In our new approach, TM can search the solution space efficiently beyond local optimality, and it stops after reaching the predefined number of iterations. In TM, three types of neighbourhood movements are defined:

1. The task T_u that is already assigned to processor P_i is picked and moved to another processor P_j, where $i! = j$.
2. The two tasks $T_u, T_v \in V^T$ that are assigned to different processors P_i and Pj are picked . The tasks are swapped, where T_u is mapped to processor P_j and T_v is mapped to processor P_i.
3. The edge $(T_u, T_v) \in E^T$ is picked, where T_u is assigned to processor P_i and T_v is assigned to processor P_j ($i = j$ is allowed). The task T_u is moved to processor $P_{i'}$ and the task T_v is moved to processor $P_{j'}$, where $i! = i'$ and $j! = j'$.

Each neighbourhood movement is a conjunction of two types of attributes: remove

task T_u from processor P_i and add task T_u to processor P_i. The movement is tabu only if all of its attributes are tabu active. The attribute is tabu active when it is placed in a tabu list. Checking whether an attribute is tabu active can be efficiently implemented and thus can reduce the necessary time to verify if the move is tabu. The conjunction of the attributes provides flexibility for choosing new movements. The first type of movement consists of two attributes: task addition and removal to/from processor P_j. There are four attributes in the second and third neighbourhood movement types: two task additions and two task removals. The number of iterations called tabu tenures w, specifies how long the attribute is tabu active. The number of tenures can be different depending on the attribute type. We assumed in TM that for the task addition the number of tab tenures is larger than for the task removal. It is easier to remove the task from the solution and satisfy the deadline requirement than add it. If there is no improvement of the current solution, the aspiration criteria are applied for the moves that start from a certain distance from the current objective function values. The distance is updated according to the improved speed of the solution. Consequently, movements that generate worse solutions are also accepted in our approach.

Algorithm 6 Task Movement

1: **procedure** Task Movement
2: Determine the objective function value z_0 of s_0.
3: Set $T(T_u, P_i) = 0$ for all $(T_u, P_i) \in V^T \times V^P$.
4: Set $z_* = z_0$ and $s_* = s_0$.
5: **for** $k = 1, ..., K$ **do**
6: Set $z_k = \infty$.
7: **for** $(T_u, P_i) \in V^T \times V^P$ **do**
8: **if** $(T_u, P_i) > k$ **then**
9: Continue with the next pair.
10: **end if**
11: Set $s = s_{k-1}$ and $s(T_u) = P_i$.
12: **if** s is infeasible **then**
13: Continue with the next pair.
14: **end if**
15: Determine the objective function value z of s.
16: **if** $z < z_k$ **then**
17: Set $\mathrm{z}_k = z$, $s_k = s$, $T_{u,k} = u$ and $P_{i,k} = P_i$.
18: **end if**
19: **end for**
20: **if** $z_k < z_*$ **then**
21: Set $z_* = z_k$, $s_* = s_k$ and $T(T_{u,i}, s_{k-1}(T_{u,k})) = k + r$.
22: **end if**
23: **end for**
24: Return s_*.
25: **end procedure**

Figure 6 shows the algorithm for the first type of movement where task T_u is moved to another processor P_j. The algorithm starts with initial solution s_0, deadline t^d, tabu list length w, and iteration limit K. To obtain both initial solution $s_0 \in S$ and deadline t^d the following main steps are performed:

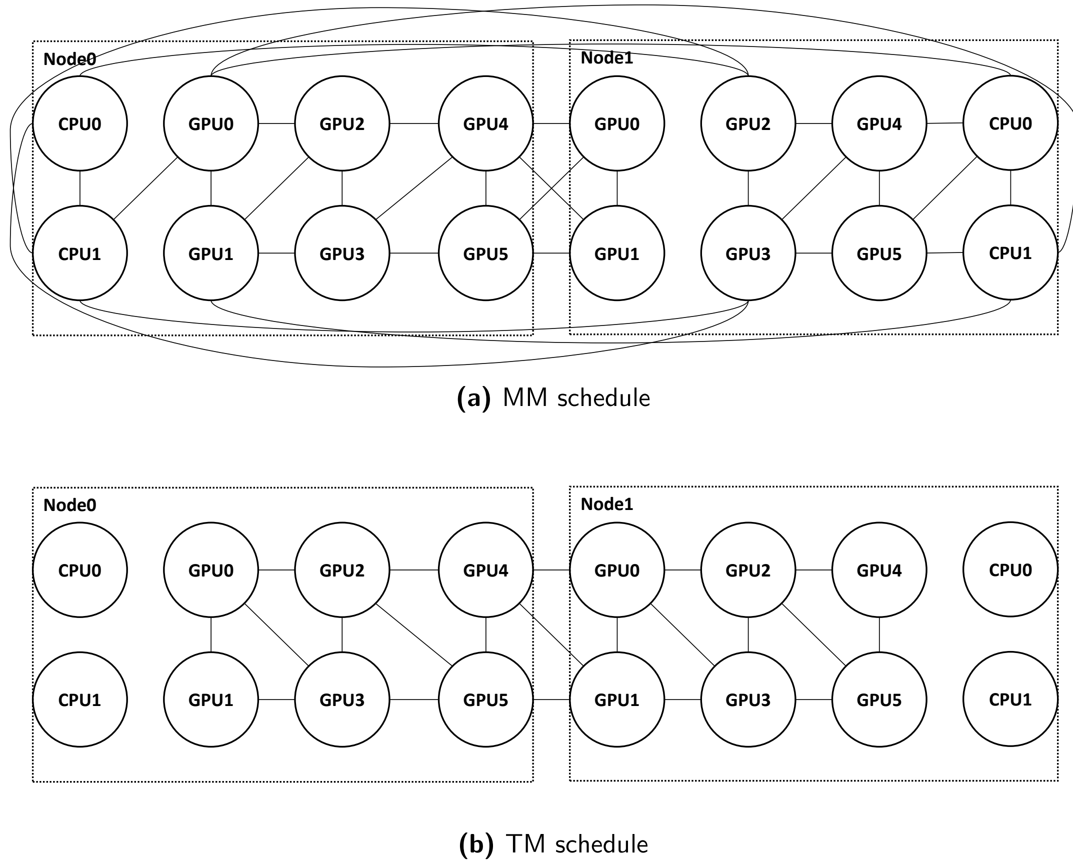
1. Generate an initial set of solutions S using the LB, DM, MM and NA heuristics.
2. Calculate stencil execution times for all the obtained solutions S.
3. Select the shortest schedule and use it as an initial deadline t^d.
4. Run the TM algorithm with each generated solution $s_0 \in S$ and find a schedule in which the total energy cost is minimized and the deadline t^d is not exceeded.

Note that TM may find a schedule with a shorter execution time than the initial deadline. Thus, we consider two objectives by reducing the energy cost and total execution time during the optimisation process in TM. Approximation of the chromatic index of the graph $\chi'(G)$ has to be calculated to determine the objective function value (Equation 5.5) of the initial solution s_0. In order to speed up the approximation calculations of the chromatic index the inequality $\chi'(G) \leq \Delta(G)+\mu(G)$ proposed in [92] was adopted, respectively.

The following new parameters need to be estimated and aggregated from the network topology and communication descriptions to apply TM for scheduling stencil computations with more realistic multi-node HPC setups:

- if tasks T_u and T_v are executed on different processors $P_i, P_j \in \mathcal{P}$, they cause the time $t^c_{(T_u,T_v),(P_i,P_j)}$ and the energy $e^c_{(T_u,T_v),(P_i,P_j)}$ penalty required to exchange the grid cells between the processors P_i and P_j. If both tasks are scheduled on the same processor, then the communication time and the communication energy are equal to zero;
- $t^c_{(T_u,T_v),(P_i,P_j)}$ the communication time depends on a communication type intranode or internode. If it is an internode communication the communication time increases with the average number of hops that a packet has to travel to reach a destination processor;
- $e^c_{(T_u,T_v),(P_i,P_j)}$ the energy cost is calculated based on the physical distance between processors and a connection type;
- the total communication time and the total communication energy to exchange all data are represented by t^c and e^c, respectively;
- t^s the total execution time indicates how much time it takes to finish the whole workload;
- t^d the execution deadline denotes the time by which all tasks have to be finished.

The energy usage of the schedule depends on the number of inter-node communications. Let us take as an example the $8x8x4$ grid with the 256 tasks distributed



(a) MM schedule

(b) TM schedule

Figure 7.1: Comparison of two example schedules and communication topologies generated by (a) MM and (b) TM algorithms for a seven-point stencil.

on two nodes (each with two CPUs and six GPUs) where the seven-point stencil is employed. Figure 7.1 shows example schedules generated by MM and TM, respectively. All the communication links between two processors are represented as a single edge to simplify the schedules. In the case of TM, CPUs are not utilised in the computations. MM and TM have 11 and 3 inter-node edges, respectively.

7.3 Computational experiments

7.3.1 Simulation setup

The problem instances were based and extracted from the real-world simulation grid used in many stencil computations, mainly numerical weather prediction (NWP) and corresponding climate models. The connection topology of points on a grid was defined by a seven-point stencil as depicted in Figure 5.8. Table 7.1 outlines the properties of the test instances. The grid called Cuboid was used to simulate decaying turbulence of a homogeneous incompressible fluid.

Let us first show the example of the decomposed Cuboid grid with connection

Table 7.1: The simulated 3D dimensional structural grid and its properties for the reference stencil-based simulations.

Name	#Blocks	#Edges	d_{T_u}	C_{T_u,T_v}	Grid size
Cuboid	8 192	46 592	64x64x64	32x32	1024x1024x1024

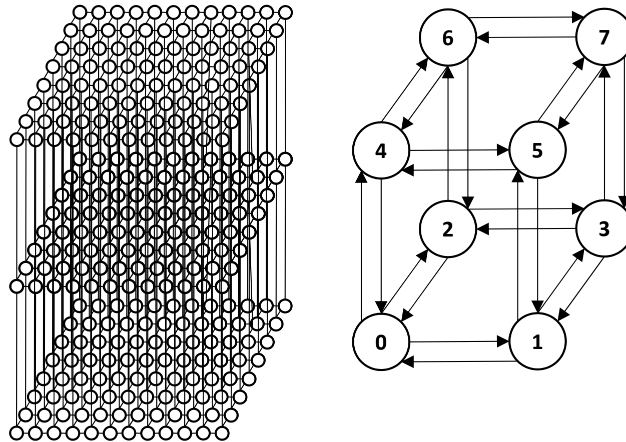


Figure 7.2: The example graph of stencil tasks with the connection dependencies.

dependencies in Figure 7.2. Each number represents the block identity that was later mapped to a specific processor. The considered grids were mapped to three different network topologies and corresponding configurations as described in Section 2.1.3 to analyse the quality of TM and simpler heuristics. All the parameters used in our experimental studies are shown in Tables 7.2, 7.3 and 7.4. The parameter values were obtained according to our methodology proposed in [26, 57]. Some extensions were naturally needed to calculate the execution time, energy usage, and communication rounds for the considered multi-node supercomputer setups and three different network topologies.

To compare three different multi-node HPC systems with different network topologies, we assumed that the overall simulated performance of each system was around 0.38 Pflop/s. Consequently, we used a different number of processors in each simulated multi-node HPC system. Table 7.2 presents three configurations of single-nodes used in the top supercomputers today. Each node consists of heterogeneous processor setups with different performance and power usage. As we can see, Nvidia Tesla V100 GPU is the highest performing parallel processor. Similarly, the best performance ratio to Thermal Design Power (TDP) is offered by Nivida Tesla V100 GPU, whereas the best ratio of the memory bandwidth to TDP is provided by ARM A64FX CPU. Arm A64FX CPU also has the highest memory bandwidth among the simulated processors. Additionally, Table 7.3 shows the intra-node and inter-node communication times and energy requirements to calculate a single stencil task. These parameters were estimated based on the technical specifications of the re-

Table 7.2: Multi-node powerful HPC systems with different processors used in experimental studies.

	Summit		Piz-Daint		Fugaku
Node	2xCPU+6xGPU		CPU+GPU		CPU
Processor	IBM Power9	Nvidia Tesla V100	Intel E5-2690v3	Nvidia Tesla P100	ARM A64FX
#Cores	22	5120	12	3584	48
Frequency [GHz]	3.07	1.3	2.6	1.19	3
Performance [Gflop/s]	540.5	7800	499	4670	2700
Memory bandwidth [GB/s]	170	900	170	732	1024
Memory size [GB]	256	16	64	16	32
Thermal Design Power (TDP) [W]	190	250	140	250	160

Table 7.3: Estimations of intra-node and inter-node communication time and energy usage in different and powerful HPC systems.

	Summit		Piz-Daint		Fugaku
Node	2xCPU+6xGPU		CPU+GPU		CPU
Processor	IBM Power9	Nvidia Tesla V100	Intel E5-2690v3	Nvidia Tesla P100	ARM A64FX
Intra-node $t^c_{(T_u,T_v),(P_i,P_j)}$ [s]	CPU-CPU $4.77*10^{-7}$	GPU-GPU $6.10*10^{-7}$	CPU-GPU $1.93*10^{-6}$		-
	CPU-GPU $6.10*10^{-7}$				
Inter-node $t^c_{(T_u,T_v),(P_i,P_j)}$ [s]	$1.22*10^{-6}$		Dragonfly $1.53*10^{-5}$ (rank 1) $1.74*10^{-5}$ (rank 2) $1.95*10^{-5}$ (rank 3)		$7.48*10^{-7}$
Intra-node $e^c_{(T_u,T_v),(P_i,P_j)}$ [J]	$3.28*10^{-5}$		$3.28*10^{-5}$		$3.28*10^{-5}$
Inter-node $e^c_{(T_u,T_v),(P_i,P_j)}$ [J]	Fat-tree $3.28*10^{-2}$ (lvl 1) $6.56*10^{-2}$ (lvl 2)		Dragonfly $3.28*10^{-2}$ (rank 1) $6.56*10^{-2}$ (rank 2)		Torus $3.28*10^{-2}*$ distance

spective top supercomputers, namely Summit and Piz-Daint. Note that each node in the Summit supercomputer has eight GPUs. Thus, there is much potential to utilise intra-node communications. According to our observation, the intra-node communication time could be even an order of magnitude lower than the inter-node communication time. The difference is even higher between the intra-node and inter-node energy usage to communicate the data. The intra-node energy usage to communicate the data is three orders of magnitude lower than the inter-node energy usage.

In the case of fat-tree network topology, we simulated two network levels where in the first network level (lvl 1) two nodes were connected, and in the second level (lvl 2) each pair of nodes was connected. Similarly, in the dragonfly network topology we had two ranks. In rank 1, the eight nodes were connected, and in rank 2, the groups of eight nodes were connected. In the case of Fugaku, as this supercomputer was in the development phase during the preparation of these experiments, all the needed parameters were estimated based on detailed descriptions discussed in [4]. The energy cost for the inter-node communication was based on the placement of nodes in the network level or rank. For the torus network we have assumed that the energy cost depends on the physical distance between nodes in the network topology. In this case, communication energy usage depends on so-called hops which measure a distance. In our experiments we simulate the torus network with 141 nodes in the $2x2x3x2x3x2$ configuration, see Section 2.1.5. Each node is directly connected to ten other nodes. However, it can simultaneously communicate data to six other nodes. To reach any node in this topology, we need a maximum of two hops so that the distance may be equal to one or two. We assumed that the energy needed to exchange data between nodes was the same in all network topologies for the corresponding network levels.

Finally, Table 7.4 presents the execution time, energy, and power usage of a single stencil task on each processor type used in our simulation experiments. As we can see in ARM A64FX, the execution time needed to process a single stencil task is the lowest among the considered processors. In comparison, Nvidia Tesla V100 has the lowest energy usage. Still, the ARM A64FX processor has lower energy usage than IBM Power9+ and Intel CPUs.

7.3.2 Simulation results

Simulation results comparing the quality of TM with simpler heuristics, introduced in previous section and in our work [26], are presented in Tables 7.5, 7.6 and 7.7. Each Table has six columns, where the second column depicts the number of three types of edges found in a schedule: all, intra-node and inter-node. Note that the number of intra-node edges does not include the edges with both tasks assigned to the same processor. The third column shows the number of processors utilised in stencil computations. The fourth column displays the summary energy cost (e^s).

Table 7.4: The estimated execution time, energy consumption, and power usage of each processor type used in simulation studies.

	Summit		Piz-Daint		Fugaku
Node	2xCPU+6xGPU		CPU+GPU		CPU
Processor	IBM Power9	Nvidia Tesla V100	Intel E5-2690v3	Nvidia Tesla P100	ARM A64FX
W_{T_u,P_i}	6	6	6	6	6
Q_{T_u,P_i}	16	16	24	16	16
$t^e_{T_u,P_i,L_u}$ [s]	$1.14 * 10^{-5}$	$2.17 * 10^{-6}$	$1.14 * 10^{-5}$	$2.67 * 10^{-6}$	$1.91 * 10^{-6}$
$e^e_{T_u,P_i,L_a}$ [J]	$5.53 * 10^{-4}$	$5.05 * 10^{-5}$	$4.41 * 10^{-4}$	$8.42 * 10^{-5}$	$9.32 * 10^{-5}$
$P^0_{T_u,P_i,L_a}$ [W]	70	70	90	70	50
$P^{idle}_{P_i}$ [W]	10	30	10	30	5
#Processors	64		148		141
Simulated performance	0.38 Pflop/s				

Table 7.5: Energy and performance achieved by TM and other heuristics for the reference grid cuboid stencil computations on the fat-tree network topology.

Heuristic	#Edges	#Processors	e^s[J]	t^d[s]	t^s[s]
Load Balancing random	61 630	64	387	-	$2.616 * 10^{-4}$
	550	16			$0.456 * 10^{-4}$
	61 080	48			$2.616 * 10^{-4}$
Load Balancing	32 448	64	139	$1.776 * 10^{-4}$	$1.776 * 10^{-4}$
	7 080	16			$0.456 * 10^{-4}$
	25 368	48			$1.776 * 10^{-4}$
Degree Minimisation	29 302	64	124	$1.776 * 10^{-4}$	$1.776 * 10^{-4}$
	6 718	16			$0.456 * 10^{-4}$
	22 584	48			$1.776 * 10^{-4}$
Multicut Minimisation	28 482	64	144	$1.776 * 10^{-4}$	$1.728 * 10^{-4}$
	5 370	16			$0.456 * 10^{-4}$
	23 112	48			$1.728 * 10^{-4}$
Neighbours Accumulation	32 448	64	140	$1.776 * 10^{-4}$	$1.776 * 10^{-4}$
	7 080	16			$0.456 * 10^{-4}$
	25 368	48			$1.776 * 10^{-4}$
Task Movement	22 728	64	104	$1.776 * 10^{-4}$	$1.650 * 10^{-4}$
	4 896	0			$1.650 * 10^{-4}$
	17 832	44			$1.560 * 10^{-4}$

Table 7.6: Energy and performance achieved by TM and other heuristics for the reference grid cuboid stencil computations on the dragonfly network topology.

Heuristic	#Edges	#Processors	e^s[J]	t^d[s]	t^s[s]
Load balancing random	1 024 284	148	485	-	$25.986 * 10^{-4}$
	456	69			$0.342 * 10^{-4}$
	1 023 828	69			$25.986 * 10^{-4}$
Load Balancing	654 934	148	257	$19.500 * 10^{-4}$	$19.500 * 10^{-4}$
	2 242	69			$0.342 * 10^{-4}$
	652 692	69			$19.500 * 10^{-4}$
Degree Minimisation	584 790	148	241	$19.500 * 10^{-4}$	$19.500 * 10^{-4}$
	4 788	69			$0.342 * 10^{-4}$
	580 002	69			$19.500 * 10^{-4}$
Multicut Minimisation	582 810	148	245	$19.500 * 10^{-4}$	$18.798 * 10^{-4}$
	4 788	69			$0.342 * 10^{-4}$
	578 022	69			$18.798 * 10^{-4}$
Neighbours Accumulation	654 934	148	257	$19.500 * 10^{-4}$	$19.500 * 10^{-4}$
	2 242	69			$0.342 * 10^{-4}$
	652 692	69			$19.500 * 10^{-4}$
Task Movement	423 532	148	173	$19.500 * 10^{-4}$	$17.735 * 10^{-4}$
	4 864	18			$1.710 * 10^{-4}$
	418 668	57			$17.735 * 10^{-4}$

The last two columns present the deadline (t^d) and the total execution time (t^s) of stencil computations. The total execution time contains the number of communication rounds (#colors) needed to exchange all data between processors, including the intra-node and inter-node communications in a reference multi-node HPC system. The stencil computations utilised in the experimental simulations were run for 1000 iterations. The execution deadline for TM was set to the shortest time achieved among simpler heuristics.

According to the obtained results, the number of communication rounds significantly affects the execution time and energy usage. Communication among processors takes more time than the execution of stencil tasks. Table 7.5 shows the obtained simulation results with the fat-tree network topology based on the Summit supercomputer. The Summit supercomputer has eight processors in each node, so called fat-nodes, and to reach the theoretical performance of 0.38 Pflop we only needed 64 processors in eight nodes). In our tests, TM reduced the number of all edges in the schedule from 28482 to 22728, whereas the number of inter-node edges was decreased by 23% compared to the MM heuristic. The number of communication rounds was also reduced from 1728 to 1596; thus, both the energy usage and the execution time were decreased by 14.1% and 4.5%, respectively, compared to the MM heuristic.

Table 7.7: Energy and performance achieved by TM and other heuristics for the reference grid cuboid stencil computations on the torus network topology.

Heuristic	#Edges	#Processors	e^s[J]	t^d[s]	t^s[s]
Load balancing random	38 360	141	347	-	$0.728 * 10^{-4}$ $0.152 * 10^{-4}$ $0.728 * 10^{-4}$
Load Balancing	25 984	141	185	$0.700 * 10^{-4}$	$0.574 * 10^{-4}$ $0.152 * 10^{-4}$ $0.574 * 10^{-4}$
Degree Minimisation	24 878	141	191	$0.700 * 10^{-4}$	$0.616 * 10^{-4}$ $0.152 * 10^{-4}$ $0.616 * 10^{-4}$
Multicut Minimisation	25 984	141	198	$0.700 * 10^{-4}$	$0.574 * 10^{-4}$ $0.152 * 10^{-4}$ $0.574 * 10^{-4}$
Neighbours Accumulation	23 408	141	181	$0.700 * 10^{-4}$	$0.574 * 10^{-4}$ $0.152 * 10^{-4}$ $0.574 * 10^{-4}$
Task Movement	24 584	114	165	$0.700 * 10^{-4}$	$0.700 * 10^{-4}$ $0.304 * 10^{-4}$ $0.700 * 10^{-4}$

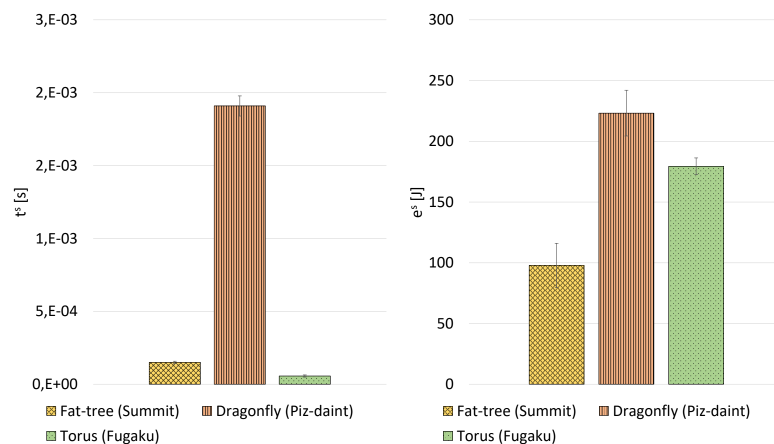


Figure 7.3: The average total execution time t^s and average summary energy cost e^s for the fat-tree, dragonfly and torus network topologies achieved by the TM algorithm.

Table 7.6 presents results for the Piz-Daint supercomputer with the dragonfly network topology. The number of communication rounds increased 10.9x times compared to the fat-tree network topology. The length of the communication round between all simulated topologies was normalised to $1x10^{-7}$s. The time needed to exchange the task data between nodes depended on the network rank or level and was up to 16x times larger on the dragonfly network topology than on the fat-tree network topology, see Table 7.3. For the dragonfly topology, TM reduced the energy cost by 28.2% and the execution time by 5.6% compared to the DM heuristic. The TM algorithm utilised only 78 processors among 148 available processors to minimise the energy usage, where 57 processors were GPUs and 18 were CPUs. The total number of edges was also decreased by 27.6% compared to the DM heuristic. The last simulated topology in our studies was the torus network implemented in the Fugaku supercomputer. In this case, TM reduced the number of edges by 16.9%, and the energy cost was reduced by 18.4%, respectively. One should note that the proposed TM algorithm successfully balanced both the communication and computation time.

Among all three major network topologies used in our simulation studies, our TM algorithm achieved $2.66x$ better performance for the torus network topology compared to the fat-tree, see Figure 7.3. However, on the second criterion - energy cost e^s - the obtained results indicated fat-tree as $1.83x$ better than the torus network topology. Consequently, we demonstrated and verified experimentally in our research how important topology-aware scheduling and task management aspects are to optimize the performance of stencil simulations in multi-node supercomputing systems.

Chapter 8
Conclusions

The primary objective of this thesis was to demonstrate that it is possible to reduce energy usage and improve the overall performance of stencil computations in a complex multi-node HPC setup. The presented research focused on three low-diameter network topologies commonly used in existing, powerful pre-exascale and exascale HPC systems.

The research contributions of this thesis can be stated as follows:

- We developed and tested experimentally the new TM algorithm to minimise both energy usage and runtime of large scale stencil computations taking into account complex fat-tree, dragonfly, and torus communication topologies and their characteristics for intra-node and inter-node setups.
- We also demonstrated experimentally how the selected communication links affect the overall performance of reference stencil computations in large-scale parallel executions.
- We showed that our new TM algorithm significantly reduces both the energy usage and the referenced stencil performance compared to previous simpler heuristics.

This work has practical applications to deal with efficient distribution of advanced stencil code executed in heterogeneous multi-node supercomputers. Based on the obtained results and simulation studies of seven-point reference stencil, we argue that the proposed scheduling model and TM algorithm can be easily extended or adapted in existing stencil application frameworks executed on any multi-node HPC setup. Our generic approach was successfully applied to all three major network topologies used in powerful HPC systems today. Moreover, our simulation analysis revealed key benefits behind fat-tree and a new, promising torus network topology. Consequently, application developers can quickly adapt and tailor the proposed simulation methodology to evaluate the efficiency of any multi-node computing systems in light of particular parallel stencil requirements, not only based on commonly used HPL or HPCG benchmarks.

In our future work we plan to compare the proposed scheduling model and sim-

ulation studies based on estimated parameters with real stencil-based computing experiments using full-scale supercomputers. We also plan to design application performance models for stencils within advanced simulator environments, such as DCworms or CODES. The presented simulation studies are useful for designing extensions in high-level stencil DSL frameworks, particularly Chemora. Moreover, the released open source software and scheduling data structures are generic, so they can also be adopted by external developers for various stencil patterns and corresponding scheduling procedures in other stencil environments, e.g. YASK or PSkel. Finally, we plan to run additional performance tests to model and predict both the energy usage and runtime for other stencil patterns and applications on constantly changing multi-node HPC architectures.

Bibliography

[1] Emmanuel Agullo, Camille Coti, Thomas Herault, Julien Langou, Sylvain Peyronnet, Ala Rezmerita, Franck Cappello, and Jack Dongarra. QCG-OMPI: MPI applications on grids. *Future Generation Computer Systems*, 27(4):357 – 369, 2011.

[2] Ashwin M. Aji, Lokendra S. Panwar, Feng Ji, Karthik Murthy, Milind Chabbi, Pavan Balaji, Keith R Bisset, James Dinan, Wu-chun Feng, John Mellor-Crummey, et al. MPI-ACC: accelerator-aware MPI for scientific applications. *IEEE Transactions on Parallel and Distributed Systems*, 27(5):1401–1414, 2015.

[3] Yuichiro Ajima and et. al. The Tofu Interconnect D. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 646–654, Sep. 2018.

[4] Yuichiro Ajima, Takahiro Kawashima, Takayuki Okamoto, Naoyuki Shida, Kouichi Hirai, Toshiyuki Shimizu, Shinya Hiramoto, Yoshiro Ikeda, Takahide Yoshikawa, Kenji Uchida, et al. The Tofu Interconnect D. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 646–654. IEEE, 2018.

[5] Sadaf R. Alam, Nicola Bianchi, Nicholas Cardo, Matteo Chesi, Miguel Gila, Stefano Gorini, Mark Klein, Marco Passerini, Carmelo Ponti, Fabio Verzelloni, and Colin McMurtrie. An Operational Perspective on a Hybrid and Heterogeneous Cray XC50 System. In *CUG2017 Proceedings*, 2017.

[6] Bob Alverson, Edwin Froese, Larry Kaplan, and Duncan Roweth. Cray xc series network. *Cray Inc., White Paper WP-Aries01-1112*, 2012.

[7] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, May 1997.

[8] Cédric Augonnet, Olivier Aumage, Nathalie Furmento, Raymond Namyst, and Samuel Thibault. StarPU-MPI: Task programming over clusters of machines enhanced with accelerators. In *European MPI Users' Group Meeting*, pages 298–299. Springer, 2012.

[9] Protonu Basu, Samuel Williams, Brian Van Straalen, Leonid Oliker, Phillip Colella, and Mary Hall. Compiler-based code generation and autotuning for geometric multigrid on GPU-accelerated supercomputers. *Parallel Computing*, 64:50 – 64, 2017. High-End Computing for Next-Generation Scientific Discovery.

[10] David E. Bernholdt, Swen Boehm, George Bosilca, Manjunath Gorentla Venkata, Ryan E. Grant, Thomas Naughton, Howard P. Pritchard, Martin Schulz, and Geoffroy R. Vallee. A survey of MPI usage in the US exascale computing project. *Concurrency and Computation: Practice and Experience*, page e4851, 2017.

[11] Kashif Bilal, Ahmad Fayyaz, Samee U. Khan, and Saeeda Usman. Power-aware resource allocation in computer clusters using dynamic threshold voltage scaling and dynamic voltage scaling: comparison and analysis. *Cluster Computing*, 18(2):865–888, 2015.

[12] Jacek Błażewicz, Klaus H Ecker, Erwin Pesch, Günter Schmidt, Malgorzata Sterna, and Jan Węglarz. *Handbook on Scheduling: from Theory to Practice*. Springer Science & Business Media, 2019.

[13] Marek Błażewicz, Steven R Brandt, Michał Kierzynka, Krzysztof Kurowski, Bogdan Ludwiczak, Jian Tao, and Jan Węglarz. Cakernel–a parallel application programming framework for heterogenous computing architectures. *Scientific Programming*, 19(4):185–197, 2011.

[14] Marek Błażewicz, Ian Hinder, David Koppelman, Steven Brandt, Miłosz Ciżnicki, Michał Kierzynka, Frank Loffler, Erik Schnetter, and Jian Tao. From physics model to results: An optimizing framework for cross-architecture code generation. *Scientific Programming*, 21(1):1–16, 2013.

[15] Béla Bollobás. *Modern graph theory*, volume 184. Springer Science & Business Media, 1998.

[16] Eric Bracken, Sergey Polstyanko, Nancy Lambert, Reiji Suda, Dennis D Giannacopulos, et al. Power aware parallel 3-D finite element mesh refinement performance modeling and analysis with CUDA/MPI on GPU and multi-core architecture. *IEEE Transactions on Magnetics*, 48(2):335–338, 2012.

[17] Victoria Caparros Cabezas and Markus Püschel. Extending the roofline model: Bottleneck analysis with microarchitectural constraints. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 222–231, 2014.

[18] Bradford L. Chamberlain, David Callahan, and Hans P. Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291–312, 2007.

[19] Aftab Ahmed Chandio, Kashif Bilal, Nikos Tziritas, Zhibin Yu, Qingshan Jiang, Samee U. Khan, and Cheng-Zhong Xu. A comparative study on resource allocation and energy efficient job scheduling strategies in large-scale parallel computing systems. *Cluster Computing*, 17(4):1349–1367, 2014.

[20] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[21] Jee Whan Choi, Daniel Bedard, Robert Fowler, and Richard Vuduc. A roofline model of energy. In *Parallel & Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, pages 661–672. IEEE, 2013.

[22] Miłosz Ciżnicki, Michał Kierzynka, Piotr Kopta, Krzysztof Kurowski, and Paweł Gepner. Benchmarking data and compute intensive applications on modern CPU and GPU architectures. In *Procedia Computer Science 9*, volume 9, pages 1900–1909, 2012.

[23] Milosz Ciżnicki, Michał Kierzynka, Piotr Kopta, Krzysztof Kurowski, and Paweł Gepner. Benchmarking JPEG 2000 implementations on modern CPU and GPU architectures. *Journal of Computational Science*, 5(2):90–98, 2014.

[24] Miłosz Ciżnicki, Michał Kulczewski, Piotr Kopta, and Krzysztof Kurowski. Methods to Load Balance a GCR Pressure Solver Using a Stencil Framework on Multi- and Many-Core Architectures. *Scientific Programming*, 2015:13, 2015.

[25] Miłosz Ciżnicki, Krzysztof Kurowski, and Jan Węglarz. Evaluation of selected resource allocation and scheduling methods in heterogeneous many-core processors and graphics processing units. *Foundations of Computing and Decision Sciences*, 39(4):233–248, 2014.

[26] Miłosz Ciżnicki, Krzysztof Kurowski, and Jan Węglarz. Energy aware scheduling model and online heuristics for stencil codes on heterogeneous computing architectures. *Cluster Computing*, 20(3):2535–2549, 2017.

[27] Miłosz Ciżnicki, Krzysztof Kurowski, and Jan Węglarz. Energy and performance improvements in stencil computations on multi-node HPC systems with different network and communication topologies. *Future Generation Computer Systems*, 115:45–58, 2021.

[28] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick.

Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 4. IEEE Press, 2008.

[29] Kaushik Datta, Samuel Williams, Vasily Volkov, Jonathan Carter, Leonid Oliker, John Shalf, and Katherine Yelick. Auto-tuning the 27-point stencil for multicore. In *In Proc. iWAPT2009: The Fourth International Workshop on Automatic Performance Tuning*, 2009.

[30] Mattias De Wael, Stefan Marr, Bruno De Fraine, Tom Van Cutsem, and Wolfgang De Meuter. Partitioned global address space languages. *ACM Computing Surveys (CSUR)*, 47(4):62, 2015.

[31] Jack J. Dongarra, Piotr Luszczek, and Antoine Petitet. The LINPACK benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

[32] Jesus Escudero-Sahuquillo, Pedro J. Garcia, Francisco J. Quiles, Sven-Arne Reinemo, Tor Skeie, Olav Lysne, and Jose Duato. A new proposal to deal with congestion in InfiniBand-based fat-trees. *Journal of Parallel and Distributed Computing*, 74(1):1802–1819, 2014.

[33] H. Feshbach and P. Morse. *Methods of Theoretical Physics*. Feshbach Publishing, 1981.

[34] Denis Foley and John Danskin. Ultra-Performance Pascal GPU and NVLink Interconnect. *IEEE Micro*, 37(2):7–17, 2017.

[35] Matteo Frigo and Volker Strumpen. Cache oblivious stencil computations. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 361–366. ACM, 2005.

[36] Michael R. Garey and David S. Johnson. Computers and intractability: a guide to the theory of NP-completeness. *WH Freeman and Company*, pages 90–91, 1979.

[37] Michael R. Garey, David S. Johnson, and Larry Stockmeyer. Some simplified NP-complete problems. In *Proceedings of the sixth annual ACM symposium on Theory of computing*, pages 47–63. ACM, 1974.

[38] Robert S. Garfinkel, George L. Nemhauser, et al. *Integer programming*, volume 4. Wiley New York, 1972.

[39] Fred Glover. Tabu search: A tutorial. *INFORMS Journal on Applied Analytics*, 20(4):74–94, 1990.

[40] Ricardo Gonzalez, Benjamin M. Gordon, and Mark A. Horowitz. Supply and threshold voltage scaling for low power CMOS. *IEEE Journal of Solid-State Circuits*, 32(8):1210–1216, 1997.

[41] William Gropp, William D Gropp, Argonne Distinguished Fellow Emeritus Ewing Lusk, Ewing Lusk, and Anthony Skjellum. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[42] Georg Hager, Jan Treibig, Johannes Habich, and Gerhard Wellein. Exploring performance and power properties of modern multi-core chips via simple machine models. *Concurrency and Computation: Practice and Experience*, pages 100–122, 2014.

[43] Marcus Hähnel, Björn Döbel, Marcus Völp, and Hermann Härtig. Measuring energy consumption for short code paths using RAPL. *SIGMETRICS Perform. Eval. Rev.*, 40(3):13–17, January 2012.

[44] Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, and Keith Underwood. Remote memory access programming in MPI-3. *ACM Transactions on Parallel Computing*, 2(2):9, 2015.

[45] Torsten Hoefler and Timo Schneider. Optimization principles for collective neighborhood communications. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–10. IEEE, 2012.

[46] Ian Holyer. The NP-completeness of edge-coloring. *SIAM Journal on Computing*, 10(4):718–720, 1981.

[47] Aleksandar Ilic, Frederico Pratas, and Leonel Sousa. Cache-aware roofline model: Upgrading the loft. *IEEE Computer Architecture Letters*, 13(1):21–24, 2014.

[48] Nikhil Jain, Abhinav Bhatele, Louis H. Howell, David Böhme, Ian Karlin, Edgar A. León, Misbah Mubarak, Noah Wolfe, Todd Gamblin, and Matthew L. Leininger. Predicting the performance impact of different fat-tree configurations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 50. ACM, 2017.

[49] Daniel Junglas. *Optimised Grid-Partitioning for Block Structured Grids in Parallel Computing*. PhD thesis, TU Darmstadt, 2007.

[50] Narendra Karmarkar. A new polynomial-time algorithm for linear programming. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 302–311. ACM, 1984.

[51] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011.

[52] John Kim, James Balfour, and William Dally. Flattened butterfly topology for on-chip networks. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 172–182. IEEE Computer Society, 2007.

[53] John Kim, Wiliam J. Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *Computer Architecture, 2008. ISCA'08. 35th International Symposium on*, pages 77–88. IEEE, 2008.

[54] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[55] Peter Kogge, S. Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, and Robert Lucas. Exascale computing study: Technology challenges in achieving exascale systems. *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Techinal Representative*, 15, 01 2008.

[56] Peter Kogge and John Shalf. Exascale computing trends: Adjusting to the" new normal" for computer architecture. *Computing in Science & Engineering*, 15(6):16–26, 2013.

[57] Krzysztof Kurowski, Miłosz Ciżnicki, and Jan Węglarz. Energy efficiency and performance modeling of stencil applications on manycore and GPU computing resources. In *2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID)*, pages 232–241. IEEE, 2020.

[58] Krzysztof Kurowski, Ariel Oleksiak, Wojciech Piątek, and Jan Węglarz. Hierarchical scheduling strategies for parallel tasks and advance reservations in grids. *Journal of Scheduling*, 16(4):349–368, 2013.

[59] Christoph Lameter. Numa (non-uniform memory access): An overview. *Queue*, 11(7):40, 2013.

[60] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[61] Dong Li, Bronis R De Supinski, Martin Schulz, Kirk Cameron, and Dimitrios S Nikolopoulos. Hybrid MPI/OpenMP power-aware computing. In *Parallel*

& Distributed Processing (IPDPS), 2010 IEEE International Symposium on, pages 1–12. IEEE, 2010.

[62] Xuan-Yi Lin, Yeh-Ching Chung, and Tai-Yi Huang. A multiple LID routing scheme for fat-tree-based InfiniBand networks. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, page 11. IEEE, 2004.

[63] Qiang Liu and Wayne Luk. *Heterogeneous Systems for Energy Efficient Scientific Computing*, pages 64–75. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[64] Kai Ma, Xue Li, Wei Chen, Chi Zhang, and Xiaorui Wang. GreenGPU: A holistic approach to energy efficiency in GPU-CPU heterogeneous architectures. In *Parallel Processing (ICPP), 2012 41st International Conference on*, pages 48–57. IEEE, 2012.

[65] Joseph S. Madachy. *Madachy's Mathematical Recreations*. Dover Publications, 1979.

[66] Tareq Malas, Georg Hager, Hatem Ltaief, and David Keyes. Towards energy efficiency and maximum computational intensity for stencil algorithms using wavefront diamond temporal blocking. *arXiv preprint arXiv:1410.5561*, 2014.

[67] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: an implicitly parallel programming model for stencil computations on large-scale GPU-accelerated supercomputers. In *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*, pages 1–12. IEEE, 2011.

[68] Jing Mei, Kenli Li, and Keqin Li. Energy-aware task scheduling in heterogeneous computing environments. *Cluster Computing*, 17(2):537–550, 2014.

[69] Paulius Micikevicius. Multi-GPU programming. *GPU Computing Webinars, NVIDIA*, 2011.

[70] Pablo D. Mininni, Duane Rosenberg, Raghu Reddy, and Annick Pouquet. A hybrid MPI–OpenMP scheme for scalable parallel pseudospectral computations for fluid turbulence. *Parallel Computing*, 37(6):316–326, 2011.

[71] Misbah Mubarak, Christopher D. Carothers, Robert Ross, and Philip Carns. Modeling a million-node dragonfly network using massively parallel discrete-event simulation. In *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, pages 366–376. IEEE, 2012.

[72] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A Ang. Introducing the Graph 500. *Cray Users Group (CUG)*, 2010.

[73] Anthony Nguyen, Nadathur Satish, Jatin Chhugani, Changkyu Kim, and Pradeep Dubey. 3.5-D blocking optimization for stencil computations on modern CPUs and GPUs. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13. IEEE Computer Society, 2010.

[74] John Nickolls, Ian Buck, Michael Garland, and Kevin Skadron. Scalable parallel programming with CUDA. *Queue*, 6(2):40–53, 2008.

[75] Lena Oden, Benjamin Klenk, and Holger Fröning. Energy-efficient Stencil Computations on Distributed GPUs Using Dynamic Parallelism and GPU-controlled Communication. In *Proceedings of the 2Nd International Workshop on Energy Efficient Supercomputing*, E2SC '14, pages 31–40, Piscataway, NJ, USA, 2014. IEEE Press.

[76] Georg Ofenbeck, Ruedi Steinmann, Victoria Caparros Cabezas, Daniele G. Spampinato, and Markus Püschel. Applying the Roofline Model. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 76 – 85, 2014.

[77] François Pellegrini. Scotch and PT-scotch graph partitioning software: an overview. *Combinatorial Scientific Computing*, pages 373–406, 2012.

[78] Alyson D Pereira, Luiz Ramos, and Luís FW Góes. PSkel: A stencil programming framework for CPU-GPU systems. *Concurrency and Computation: Practice and Experience*, 2015.

[79] Fabrizio Petrini and Marco Vanneschi. k-ary n-trees: High performance networks for massively parallel architectures. In *Proceedings 11th International Parallel Processing Symposium*, pages 87–93. IEEE, 1997.

[80] Judit Planas, Rosa M Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with StarSs. *The International Journal of High Performance Computing Applications*, 23(3):284–299, 2009.

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

[82] Joseph M. Prusa, Piotr K. Smolarkiewicz, and Andrzej A. Wyszogrodzki. EULAG, a computational model for multiscale flows. *Computers & Fluids*, 37(9):1193–1207, 2008.

[83] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core SMP nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[84] Daniel A. Reed and Jack Dongarra. Exascale computing and big data. *Communications of the ACM*, 58(7):56–68, 2015.

[85] Krzysztof Andrzej Rojek, Miłosz Ciżnicki, Bogdan Rosa, Piotr Kopta, Michał Kulczewski, Krzysztof Kurowski, Zbigniew Paweł Piotrowski, Łukasz Szustak, Damian Karol Wójcik, and Roman Wyrzykowski. Adaptation of fluid model EULAG to graphics processing unit architecture. *Concurrency and Computation: Practice and Experience*, 27(4):937–957, 2015.

[86] Subhash Saini, Johnny Chang, and Haoqiang Jin. Performance evaluation of the Intel Sandy Bridge based NASA pleiades using scientific and engineering applications. In *International Workshop on Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems*, pages 25–51. Springer, 2013.

[87] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14(3):219–240, 2002.

[88] Erik Schnetter, Marek Błażewicz, Steven R. Brandt, David M. Koppelman, and Frank Löffler. Chemora: a PDE-solving framework for modern high-performance computing architectures. *Computing in Science & Engineering*, 17(2):53–64, 2015.

[89] Alexander Schrijver. *Theory of linear and integer programming.* John Wiley & Sons, 1998.

[90] Sriram Sellappa and Siddhartha Chatterjee. Cache-efficient multigrid algorithms. *International Journal of High Performance Computing Applications*, 18(1):115–133, 2004.

[91] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. In *High Performance Computing for Computational Science–VECPAR 2010*, pages 1–25. Springer, 2011.

[92] Claude E Shannon. A theorem on coloring the lines of a network. *Journal of Mathematics and Physics*, 28(1):148–152, 1949.

[93] William M. Spears, Kenneth A. De Jong, Thomas Bäck, David B. Fogel, and Hugo de Garis. An overview of evolutionary computation. In *ECML*, 1993.

[94] Data structures with problem instances repository by Miłosz Ciżnicki website. github, 2022. online: https://github.com/miloszc/task-movement.

[95] Hamdy A. Taha. *Integer programming: theory, applications, and computations*. Academic Press, 2014.

[96] George Terzopoulos and Helen D. Karatza. Power-aware bag-of-tasks scheduling on heterogeneous platforms. *Cluster Computing*, 19(2):615–631, 2016.

[97] J. Treibig, G. Hager, and G. Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In *Proceedings of PSTI2010, the First International Workshop on Parallel Software Tools and Tool Infrastructures*, San Diego CA, 2010.

[98] Jan Treibig and Georg Hager. Introducing a performance model for bandwidth-limited loop kernels. In *Parallel Processing and Applied Mathematics*, pages 615–624. Springer, 2010.

[99] Jan Treibig, Gerhard Wellein, and Georg Hager. Efficient multicore-aware parallelization strategies for iterative stencil computations. *Journal of Computational Science*, 2(2):130–137, 2011.

[100] Didem Unat, Xing Cai, and Scott B. Baden. Mint: realizing CUDA performance in 3D stencil methods with annotated C. In *Proceedings of the international conference on Supercomputing*, pages 214–224. ACM, 2011.

[101] Oreste Villa, Daniel R Johnson, Mike Oconnor, Evgeny Bolotin, David Nellans, Justin Luitjens, Nikolai Sakharnykh, Peng Wang, Paulius Micikevicius, Anthony Scudiero, et al. Scaling the power wall: a path to exascale. In *High Performance Computing, Networking, Storage and Analysis, SC14: International Conference for*, pages 830–841. IEEE, 2014.

[102] Vadim G. Vizing. The chromatic class of a multigraph. *Cybernetics and Systems Analysis*, 1(3):32–41, 1965.

[103] Po-Han Wang, Chia-Lin Yang, Yen-Ming Chen, and Yu-Jung Cheng. Power gating strategies on GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(3):13, 2011.

[104] Zhuowei Wang, Xianbin Xu, Naixue Xiong, Laurence T. Yang, and Wuqing Zhao. Energy cost evaluation of parallel algorithms for multiprocessor systems. *Cluster Computing*, 16(1):77–90, 2013.

[105] Counting Floating Point Operations website, 2019. http://icl.cs.utk.edu/projects/papi/wiki/PAPITopics:SandyFlops.

[106] Green500 List website, 2022. online: https://www.top500.org/green500/list/2017/06/.

[107] Intel Advisor website, 2022. online: https://software.intel.com/en-us/intel-advisor-xe.

[108] Nvidia Management Library website, 2017. online: https://developer.nvidia.com/nvidia-management-library-nvml.

[109] Nvidia Profiler website, 2017. online: https://developer.nvidia.com/nvidia-visual-profiler.

[110] Top500 List website, 2022. online: http://top500.org.

[111] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[112] Noah Wolfe, Misbah Mubarak, Christopher D. Carothers, Robert B. Ross, and Philip H. Carns. Modeling large-scale slim fly networks using parallel discrete-event simulation. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 28(4):29, 2018.

[113] Noah Wolfe, Misbah Mubarak, Nikhil Jain, Jens Domke, Abhinav Bhatele, Christopher D. Carothers, and Robert B. Ross. Preliminary performance analysis of multi-rail fat-tree networks. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 258–261. IEEE, 2017.

[114] Wei Xue, Chao Yang, Haohuan Fu, Xinliang Wang, Yangtong Xu, Lin Gan, Yutong Lu, and Xiaoqian Zhu. Enabling and scaling a global shallow-water atmospheric model on Tianhe-2. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pages 745–754. IEEE, 2014.

[115] Charles Yount, Alejandro Duran, and Josh Tobin. Multi-level spatial and temporal tiling for efficient HPC stencil computation on many-core processors with large shared caches. *Future Generation Computer Systems*, 92:903 – 919, 2019.

[116] Charles Yount, Josh Tobin, Alexander Breuer, and Alejandro Duran. YASK—Yet another stencil kernel: A framework for HPC stencil code-generation and tuning. In *2016 Sixth International Workshop on Domain-Specific Languages and High-Level Frameworks for High Performance Computing (WOLFHPC)*, pages 30–39. IEEE, 2016.

[117] Yun Zou and Sanjay Rajopadhye. Automatic energy efficient parallelization of uniform dependence computations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, pages 373–382, New York, NY, USA, 2015. ACM.

Appendix A

Software and data repository

To support the computational reproducibility of presented results and stencil benchmarks we have created a dedicated repository to share our data structures, algorithms, and problem instances together with guidelines for running and comparing computational experiments in [94]. To implement TM, we have adopted the KOALA graph partitioning software. KOALA is a basic set of C++ templates supporting various required functionalities in the fields of algorithmic graph theory and network problems in discrete optimisation. Many well-established libraries are supporting a variety of different graph partitioning schemes, particularly Metis and its extended versions [87] or SCOTCH [77]. However, the KOALA enables both easy implementation and computation of relevant properties of the multigraph, such as maximum degree and maximum number of parallel edges. KOALA also supports the *.graphml file format, which is the commonly adopted standard representation used for the graph data exchange.

We provided readers and stencil application developers with open access to all the experimental data for better understanding of the proposed TS algorithm. Our software together with reference research data enable researchers to easily run, check and reuse our experimental data stencil structures, algorithms and problem instances for further comparisons and improvements. Using our reference software repository, one can adopt the scheduling model and reuse any of the heuristic methods, including TM. One can easily link our libraries with existing parallel execution environments for efficient stencil code executions on multi-node supercomputers.

List of Figures

2.1 Architecture of the CPU (left) and the GPU (right) 15
2.2 Two CPUs connected with two QPI links 17
2.3 Four CPUs connected with four QPI links 17
2.4 Root complex with a PIC Express topology 18
2.5 Root complex with four GPUs 18
2.6 Two GPUs connected to two separate CPUs 18
2.7 NVLink connetcing four GPUs 19
2.8 Fat-tree network topology 20
2.9 Fat-tree network topology implemented in the IBM Power 9+ Summit multi-node HPC system. 21
2.10 Torus network topology with different dimensions 22
2.11 Torus network topology implemented in K-computer and Post-K supercomputer. 23
2.12 Dragonfly network topology 24
2.13 Dragonfly network topology implemented in Cray XC50 Piz-Daint multi-node HPC system. 25

3.1 Seven-point stencil 33
3.2 Twenty-seven point stencil 34
3.3 The example 3D grid where the red colour shows the interior and the yellow colour shows boundaries 35
3.4 The generic view of the 2.5D blocking method 37
3.5 Subgrid with the halo region 38
3.6 Architecture with a two-level memory hierarchy 40
3.7 The Roofline model 41
3.8 Four different computation types on the Roofline model diagram . . . 42
3.9 Roofline model of an example application 42
3.10 Cache aware Roofline model 43
3.11 Cache-aware Roofline model of an example application 44
3.12 Roofline model of energy 46

5.1 Performance of a seven-point stencil: left - the CPU, right - the GPU 58
5.2 Memory bandwidth benchmark on: left - the CPU, right - the GPU . 59
5.3 Energy usage of a seven-point stencil on the CPU 59
5.4 Power consumption of a seven-point stencil: left - the CPU, right - the GPU 61
5.5 PKG, core and DRAM power consumption of a seven-point stencil on the CPU 61
5.6 Power consumption of a seven-point stencil on the CPU 62
5.7 Performance to energy cost per lattice update for a seven-point stencil using different domain sizes on the CPU and the GPU 62
5.8 seven-point reference stencil communication pattern. 63
5.9 Write allocate in Intel CPUs 67
5.10 Constant power P0: left - the CPU, right - the GPU 72

6.1 Cuboid 81
6.2 Sphere 81
6.3 Graph of stencil tasks with the connection dependencies 81
6.4 Graph of processors 82
6.5 Comparison of schedule between *Alg_1* and *Alg_4*. The colours represent the scheduling of blocks to the processors: red - GPU00, blue - GPU01 and green - CPU00. Left - output from *Alg_1*, right - output from *Alg_4*. 88
6.6 Comparison of accuracy (%) between the proposed model and the real measurements on the Intel Xeon E5-2670@2.6GHz processor. . . 89
6.7 Comparison of accuracy (%) between the proposed model and the real measurements on the Nvidia K20m accelerator. 89

7.1 Comparison of two example schedules and communication topologies generated by (a) MM and (b) TM algorithms for a seven-point stencil. 95
7.2 The example graph of stencil tasks with the connection dependencies. 96
7.3 The average total execution time t^s and average summary energy cost e^s for the fat-tree, dragonfly and torus network topologies achieved by the TM algorithm. 101

List of Tables

2.1 Energy consumption of processing unit components 14
2.2 Top500 list from November 2021 of most powerful multi-node HPC systems ranked according to the High-Performance Linpack (HPL) and High Performance Conjugate Gradients (HPCG) benchmarks. . . 20

5.1 Properties of the modern architectures. The GPU bandwidth with Error Correcting Code (ECC) switched on/off. 58
5.2 Execution time for the CPU and GPU architectures based on the code analysis. 69
5.3 W_{T_u,P_i} for different compilers and flags on the CPU. 70
5.4 Q_{T_u,P_i} for different compilers and flags on the CPU. 70
5.5 Generated histogram for ICC compiler. 71
5.6 Energy coefficients for the CPU and GPU architectures. 71

6.1 Number of variables and constraints that formulate the ILP problem. 76
6.2 Properties of the simulated grids. 81
6.3 Parameters setup. 82
6.4 ILP on Cuboid with all configurations. 83
6.5 ILP on Sphere with all configurations. 83
6.6 Heuristics on Cuboid with the CPU-CPU configuration. 84
6.7 Heuristics on Cuboid with the CPU-GPU configuration. 84
6.8 Heuristics on Cuboid with the 2xCPU-2xGPU configuration. 85
6.9 Heuristics on Sphere with the CPU-CPU configuration. 85
6.10 Heuristics on Sphere with the CPU-GPU configuration. 86
6.11 Heuristics on Sphere with the 2xCPU-2xGPU configuration. 86
6.12 The average execution time (us) of the investigated ILP model and heuristics for the 2xCPU-2xGPU configuration. 88
6.13 Energy usage (J) of the proposed model and the real measurements for the investigated ILP model and heuristics. 89
6.14 Comparison of accuracy (%) between the proposed model and the real measurements for the investigated ILP model and heuristics. . . 90

7.1 The simulated 3D dimensional structural grid and its properties for the reference stencil-based simulations. 96
7.2 Multi-node powerful HPC systems with different processors used in experimental studies. 97
7.3 Estimations of intra-node and inter-node communication time and energy usage in different and powerful HPC systems. 97
7.4 The estimated execution time, energy consumption, and power usage of each processor type used in simulation studies. 99
7.5 Energy and performance achieved by TM and other heuristics for the reference grid cuboid stencil computations on the fat-tree network topology. 99
7.6 Energy and performance achieved by TM and other heuristics for the reference grid cuboid stencil computations on the dragonfly network topology. 100
7.7 Energy and performance achieved by TM and other heuristics for the reference grid cuboid stencil computations on the torus network topology. 101