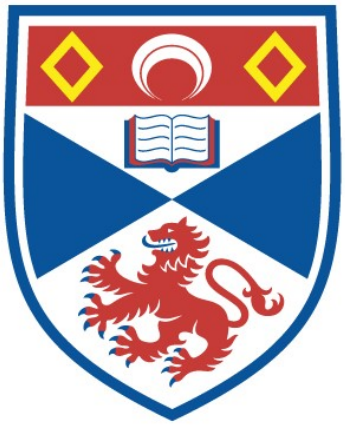
HETEROGENEITY-AWARE SCHEDULING AND DATA PARTITIONING FOR SYSTEM PERFORMANCE ACCELERATION

Teng Yu

A Thesis Submitted for the Degree of PhD
at the
University of St Andrews



2020

Heterogeneity-Aware Scheduling and Data Partitioning for System Performance Acceleration

Teng Yu

This thesis is submitted in partial fulfilment for the degree of

Doctor of Philosophy (PhD)

at the University of St Andrews

October 2019

Candidate's declaration

I, Teng Yu, do hereby certify that this thesis, submitted for the degree of PhD, which is approximately 80,000 words in length, has been written by me, and that it is the record of work carried out by me, or principally by myself in collaboration with others as acknowledged, and that it has not been submitted in any previous application for any degree.

I was admitted as a research student at the University of St Andrews in September 2016.

I received funding from an organisation or institution and have acknowledged the funder(s) in the full text of my thesis.

Date 2020.03.20 Signature of candidate

Supervisor's declaration

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of PhD in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree.

Date 2020.03.20 Signature of supervisor

Permission for publication

In submitting this thesis to the University of St Andrews we understand that we are giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. We also understand, unless exempt by an award of an embargo as requested below, that the title and the abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker, that this thesis will be electronically accessible for personal or research use and that the library has the right to migrate this thesis into new electronic forms as required to ensure continued access to the thesis.

I, Teng Yu, confirm that my thesis does not contain any third-party material that requires copyright clearance.

The following is an agreed request by candidate and supervisor regarding the publication of this thesis:

Printed copy

No embargo on print copy.

Electronic copy

No embargo on electronic copy.

Date 2020.03.20 Signature of candidate

Date 2020.03.20 Signature of supervisor

Underpinning Research Data or Digital Outputs

Candidate's declaration

I, Teng Yu, hereby certify that no requirements to deposit original research data or digital outputs apply to this thesis and that, where appropriate, secondary data used have been referenced in the full text of my thesis.

Date 2020.03.20 Signature of candidate

Abstract

Over the past decade, heterogeneous processors and accelerators have become increasingly prevalent in modern computing systems. Compared with previous homogeneous parallel machines, the hardware heterogeneity in modern systems provides new opportunities and challenges for performance acceleration. Classic operating systems optimisation problems such as task scheduling, and application-specific optimisation techniques such as the adaptive data partitioning of parallel algorithms, are both required to work together to address hardware heterogeneity.

Significant effort has been invested in this problem, but either focuses on a specific type of heterogeneous systems or algorithm, or a high-level framework without insight into the difference in heterogeneity between different types of system. A general software framework is required, which can not only be adapted to multiple types of systems and workloads, but is also equipped with the techniques to address a variety of hardware heterogeneity.

This thesis presents approaches to design general heterogeneity-aware software frameworks for system performance acceleration. It covers a wide variety of systems, including an OS scheduler targeting on-chip asymmetric multi-core processors (AMPs) on mobile devices, a hierarchical many-core supercomputer and multi-FPGA systems for high performance computing (HPC) centers. Considering heterogeneity from on-chip AMPs, such as thread criticality, core sensitivity, and relative fairness, it suggests a collaborative based approach to co-design the task selector and core allocator on OS scheduler. Considering the typical sources of heterogeneity in HPC systems, such as the memory hierarchy, bandwidth limitations and asymmetric physical connection, it proposes an application-specific automatic data partitioning method for a modern supercomputer, and a topological-ranking heuristic based schedule for a multi-FPGA based reconfigurable cluster.

Experiments on both a full system simulator (GEM5) and real systems (Sunway Taihulight Supercomputer and Xilinx Multi-FPGA based clusters) demonstrate the significant advantages of the suggested approaches compared against the state-of-the-art on variety of workloads.

Acknowledgements

Sincere thanks to Dr. John D. Thomson for his supervision and countless help during my PhD period in St Andrews. It is my pleasure to work with such an excellent supervisor and friend like John. Under his supervision, I can fully enjoy the truly beauty with freedom by conducting original research and keep on the right track. When I get problems, he will come to help in time; When I deep in troubles, he never give me up; Whenever I need backup, he tells me: *I'm here*. I will never forget the moment before a conference deadline - I got wrong of the exact submission time and just told him one hour before the deadline at 11:30 pm. John replied me on Skype within 5 minutes and told me: *I'm working on it now*. During the myriad days of risky PhD research, He carries me and stays with me. To be honest, all my family, friends and our research collaborators in China name him as *Da Zhuang* - (the Chinese pronunciation of The Great John).

Warm thanks to Prof. Hugh Leather and Dr. Pavlos Petoumenos in University of Edinburgh. Their efficient advises and technical hints support me to finally get here - appreciate for all their helps and thanks to never give me up. Special thanks to Dr. Wenlai Zhao, Bo Feng, Prof. Haohuan Fu, Prof. Yuchun Ma, Dr. Liandeng Li, Dr. Weijie Zheng and Pan Liu - the wonderful collaborators from Tsinghua University and National Supercomputer Centre in Wuxi, China. The discussion and collaborated work with their group motivates me a lot and leads me to the novel research areas on high performance computing and work on the world-leading Sunway Taihulight supercomputer.

My thanks to the following friends for their helps and suggestions during my PhD period: Dr. Vladimir Janjic (University of St Andrews), Xia Zhao (Ghent University), Min Li (Chinese Academy of Science), Mingcan

Zhu (University of Edinburgh), Xiaohan Yan (University of California, Berkeley), Dr. Shicai Wang (Imperial College London), Dr. Yuhui Lin (University of St Andrews), Peng Sun (University of Cambridge), Bozhao Li (Google), Dr. Mark Stillwell (Apple), Prof. Michel Schellekens (University College Cork), Bojia Ma (City University of Hong Kong), Yanbei Chen (University of London), Dr. Oleksandr Murashko (University of St Andrews), Nicolas Zurbuchen (University of St Andrews) and Xue Guo (University of St Andrews).

Finally, appreciate to my parents and my little faery, *Chenyu*. During the myriad silence days, you stay with me. Whenever I feel tired, disappointed or dispirited, you hold me tight. Loneliness and hopelessness leave me away with you hand in hand by my side - *With a faery, hand in hand. For the world's more full of weeping than we can understand (W.B. Yeats).*

This work is supported by *St Leonards 7th Century Scholarship* and *Computer Science PhD funding* from University of St Andrews; by UK EPSRC grant *Discovery: Pattern Discovery and Program Shaping for Manycore Systems* (EP/P020631/1).

Teng Yu
St Andrews
Scotland, UK
March 9, 2020

To My Parents Yun Zhou & Wenbin Yu

To My Lover Chenyu Wang

To My Son Zichen Yu

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Efficient task scheduling is central to improving system performance on modern heterogeneous architectures.

Heterogeneity-aware techniques have been widely researched and applied to modern computer systems, from high performance computing (HPC) and supercomputers to embedded mobile devices. High quality schedulers are designed to consider multiple objectives in their optimisation, with heterogeneity coming from both the hardware design and the workload composition. However, such schedulers are still limited in their scope, either to particular configurations of hardware heterogeneity or to particular software asymmetries.

This thesis presents a general design principle to address the system heterogeneity issues and leads to a set of collaborative approaches to schedule programs and partition data. These include the COLAB scheduler for asymmetric chip multi-core processors, the SupCOLAB scheduler for large-scale hierarchical many-core processors and the AccCOLAB for FPGA-based multi-accelerator clusters.

1.1 Motivation

In embedded mobile devices, such as smartphones and IoT sensors, energy and power constraints are central in designing new processors. The power wall limits how much switching activity can occur on each chip. Heterogeneous systems are necessary under such constraints, providing energy-efficient processing for different types of workloads [89].

Single Instruction Set Architecture (ISA) based asymmetric multicore processors (AMP) introduce new opportunities and challenges [74]. Since all processors share the same ISA, OS schedulers do not have to prematurely tie a program's implementation to a specific type of processor. Instead they can make decisions at runtime, based not only on which processor is appropriate for the workload but also on which processors are under-utilized. This introduces an extra degree of freedom to the already complex scheduling decision space, making an already challenging problem much harder.

In addition, the hierarchical multi-core CPUs used in HPC need to be scheduled carefully to fit application-specific requirements. Large-scale HPC-oriented algorithms are often custom-designed to fit the unique advantages and idiosyncrasies of the particular hierarchical memory architecture on a parallel machine. Consequently, mapping the software asymmetry smartly to the hardware heterogeneity opens a new door for further performance gain when solving scientific problems on large-scale supercomputers. This includes not only scheduling of parallel workflows to match hardware heterogeneity [40, 82, 18], but also efficient self-aware partitioning [65, 9] of dataflow to fit the hierarchical hardware composition.

Beyond the general purpose CPUs, accelerators are becoming increasingly prevalent in heterogeneous systems [29, 26]. FPGAs have been shown to be fast and power efficient for particular tasks, yet scheduling on FPGA-based multi-accelerator systems is challenging and relatively poorly researched problem, where workloads can vary significantly in granularity in terms of task size and/or number of computational units required. Schedulers need to dynamically make decisions on networked multi-FPGA systems while maintaining high performance, even in the presence of irregular workloads.

1.2 Problem Models

This section presents the underlying problem models and representations for scheduling and hierarchical data partitioning, targeting modern heterogeneous systems.

1.2.1 Problem Representation of Scheduling

The general underlying problem of scheduling can be represented as a bin-packing problem, where there are a set of bins (*hardware resources*) and a

collection of balls (*software programs*). The target is always to allocate all balls to bins satisfying specific goals. Formalized, given n task samples:

$$\mathcal{T} = \{t_i \mid i \in \{1, ..., n\}\}$$

and m hardware resources:

$$\mathcal{R} = \{r_j \mid j \in \{j, ..., m\}\}$$

the goal is to find a map f from $\mathcal{T}$ to $\mathcal{R}$:

$$f : \mathcal{T} \to \mathcal{R}$$

which can achieve either a specific or multi-objective goal, such as high performance (maximum throughput, minimum turnaround time) or a tradeoff between performance and energy. This underlying theoretical bin-packing problem has been fully studied in the literature and demonstrated to be NP-hard for obtaining the optimal solution [107]. Heuristic-based approaches have also been widely studied to provide efficient O(n) solutions of bin-packing [78]. While targeting the real task scheduling problem extended from the theoretical model, the main difficulties comes from the difference between each task t_i and each resource r_j together with the complex relationships between them.

In detail, the traditional difficulties of task scheduling on homogeneous computing systems come mainly from the complexity of the set of tasks $\mathcal{T}$. Each task can be different in size and load. For instance, different parallel processes from the same pipelined pthread program will need to execute different code segments in the same time. While even for a simple data-parallel OpenMP program, different thread might need to address different portion of the input dataflow, which leads to load imbalance between parallel processors. Tasks can additionally have different arrival times and user requirements. Further, parallel tasks can have data dependences, which gives them additional constraints on order of execution.

Modern heterogeneous systems bring new challenges and difficulties to task scheduling based on the heterogeneities from hardware resources $\mathcal{R}$. Even on a fully CPU based system without any specialised hardware accelerators, multiple on chip cores can differ in computing ability, including instruction execution model (in order or out of order), frequency and the energy consumption ratio. Threads to be scheduled on these asymmetric cores will

thus have different behaviour. Further, programs with multiple co-executed threads suffer more difficulty in achieving fairness between parallel execution on asymmetric cores. The multi-core or manycore processors also bring heterogeneities from architectural hierarchy. For instance, there is a huge gap in communication speed between on-chip multicore processors and networked computing nodes, together with the storage limitation between each core's associated cache and the shared memory. Finally, tasks will be executed under different models on heterogeneous hardware resources, including the *go in once* way such as how typical programming models (OpenCL, CUDA) for GPUs and FPGAs operate, and the time-slice based equal progress model which mainly operate on modern multitasking operating systems (Linux) for CPUs.

1.2.2 Problem Representation of Hierarchical Data Partitioning

Hierarchical data partitioning is a common approach to achieve parallel computing and improve performance. It is important for executing scientific workloads on high performance computing systems (HPC), for which the specific parallel algorithm can be co-designed to fit the hardware hierarchy. Basic data partitioning aims to partition the original dataflow into multiple symmetric portions to execute in parallel on multiple cores.

Formalized, given n data samples:

$$\mathcal{D} = \{d_i \mid i \in \{1, ..., n\}\}$$

and m parallel computing resources:

$$\mathcal{R} = \{r_j \mid j \in \{j, ..., m\}\}$$

the goal is to find a partitioning P in $\mathcal{D}$, for which $\mathcal{D}$ is divided into m portions and then be allocated to each element of $\mathcal{R}$. In this case, each hardware resource only needs to address $\frac{n}{m}$ of the original dataflow.

Further, hierarchical partitioning is designed to address large-scale dataflows. It groups the hardware resources into resource groups $\{\mathcal{R}\}$ first, either based on physical architecture or virtual machine:

$$\{\mathcal{R}\}_{j'} = \{r_j \mid \ j \in (1 + (j' - 1) * m',\ j' * m')\}$$

where j' is the index of each resource group and m' denotes the amount of resources in each group. In this case, each group needs to address $\frac{n}{m} * m'$ of the original dataflow. Further, consider that each data sample d_i is composed by k dimensions:

$$d_i = \{d_{il} \mid l \in \{1, ..., k\}\}$$

These dimensions can then be further partitioned into multiple resources inside each group, so each resource only needs to handle partial dimensions $(\frac{k}{m'})$ of each data sample d.

The priciple difficulty in achieving efficient hierarchical data partitioning is that, given the same amount of hardware resources m, hierarchical data partitioning can not directly reduce the load of each thread, but might generate additional synchronous overhead:

$$(\frac{n}{m} * m') * \frac{k}{m'} == \frac{n}{m} * k$$

Note that the right hand side of the formula above is the load of each thread under basic partitioning, as it needs to address the whole data sample with k dimensions.

Heterogeneous hardware provides the opportunity for efficient hierarchical data partitioning. For instance, if the given resources, m, are organised hierarchically with different physical connections, then hierarchical partitioning can provide the mapping between the asymmetric synchronisation needs from parallel threads onto the different communication speeds from hierarchical resources. Further, under hierarchical data partitioning, each resource only needs to read and address partial data samples during runtime, which reduces the need of storage space. Hierarchical partitioning provides the opportunity to handle large-scale workloads where each data sample is too large to be stored in cache without further partitioning.

1.3 Research Questions

This section presents the underlying research questions discussed in this thesis. It first describes the general research questions based on the problem representations above. It then describes the design of efficient heterogeneity-aware schedulers across different scales of heterogeneous systems – from asymmetric chip multi-core processors to hierarchical many-core processors and FPGA-based multi-accelerator clusters.

1.3.1 General Research Problems

The general research question for scheduling, following from the problem representation above is: *how to map the software asymmetry to hardware heterogeneity efficiently.* In detail, we must take into consideration multiple factors, such as core sensitivity, bottlenecks and fairness, which all influence the decision. In order to answer this question, the scheduler is expected to address all these factors in a collaborative way.

The general question for hierarchical data partitioning can be stated as: *how to use the asymmetry of parallel tasks from given workloads to fit the hardware heterogeneity.* The HPC-oriented workloads are expected to be customised to exploit the specific hardware advantages of a system.

1.3.2 Scheduler for Asymmetric Multi-core Processors

Considering asymmetric chip multi-core processors for mobile devices, the three research problems influencing the decisions of a general purpose multi-core scheduler are:

- Each core type is designed to optimally handle different kinds of workloads. For example, in ARM big.LITTLE systems, big cores are designed to serve latency-critical workloads or workloads with Instruction Level Parallelism (ILP). Running other kinds of workloads on them would not improve performance significantly while consuming more energy. Predicting which threads would benefit the most from running on each kind of core is critical to build an efficient AMP scheduler.

- Executing a particular thread faster does not necessarily translate into improved performance overall. If the threads of the application are unbalanced or are executed at different speeds, e.g. because an AMP is used and different threads run on different types of cores, the application will be only as fast as the most critical thread. Based on the well-known *Amdahl's Law*, the best schedule would likely prefer the most critical thread as much as possible, regardless of core sensitivity.

- In multi-programmed environments, making decisions to accelerate each application in isolation is not enough. Decisions should not only improve the utilization of the system as whole, but should not penalize any application disproportionately. Ideally, resource sharing equally across

all applications results in negative impact while fairness is the real target. For traditional schedulers this was relatively straightforward: give applications CPU slots of equal time in a round robin manner or by red-black tree. AMPs make this simple solution unworkable. The same amount of CPU time results in completely different and varying amounts of work on different heterogeneous processing elements.

1.3.3 Scheduler for Hierarchical Many-core Supercomputers

Considering hierarchical many-core supercomputers, the scheduler and data partitioner should consider both the hierarchical hardware architecture and customized parallel algorithms when optimising. Issues include:

- The data dependency and communication-to-computation ratio between each parallel thread of the given algorithm.
- The trade-off between partitioning the data structure and the processed size and number of dimensions for the datasets.
- The hierarchical heterogeneity from the hardware resources, including the ability of each computational unit, the accessible storage size and the bandwidth/ communication speed between different units.

1.3.4 Scheduler for FPGA-based Multi-accelerator Clusters

Dynamically scheduling irregular workloads in a FPGA-based multi-accelerator system is a complex task. An adequate solution should not only efficiently schedule multiple irregular tasks, but also assigns corresponding resources to hardware accelerators during runtime. This is difficult to achieve in practice:

- Multiple tasks should be executed quickly and simultaneously to achieve a short total execution time and high throughput.
- Tasks can be instantiated on different system configurations with differing numbers of accelerators, so an ideal configuration cannot be easily determined a priory.
- Tasks require different topologies between hardware accelerators based on what communication is needed during runtime.

1.4 Research Methods

This section briefly illustrates the heterogeneity-aware collaborative methods proposed in this thesis to tackle the above research problems:

1.4.1 COLAB: A Collaborative Multi-factor Scheduler for Asymmetric Multi-core Processors

A novel collaborative approach is suggested to consider all three main scheduling factors (core sensitivity, thread criticality and fairness) simultaneously.

This approach involves using an off-line trained machine learning model to assign each thread to heterogeneous CPU cores based on predicated speedup. It achieves multiple bottleneck co-acceleration on both big and little cores by dynamic task reordering and preemption, which handles the influence from massive multi-threaded multi-programmed workloads. It also scales time slicing for each thread on big cores to achieve fairness, represented by equal-progress. These three techniques work together to produce an improved scheduler for handling mixed multi-programmed workloads.

The proposed collaborative scheduler [111, 112] has been evaluated on a well-known simulator with a variety of hardware configurations targeting mixed multi-threaded multi-programmed workloads. It outperforms the default Linux scheduler and the state-of-the-art heterogeneity-aware scheduler, both in terms of turnaround time and system throughput.

1.4.2 SupCOLAB: A Collaborative Scheduler with Data Partitioning for Many-core Supercomputers

The heterogeneity and customized large-scale workload on a modern supercomputer is addressed by developing a collaborative scheduler, equipped with an automatic data partitioner, targeting the given hierarchical hardware resources with many-core processors.

Using this approach, the large-scale workflow will first be auto-partitioned into different parallel tasks based on the input data structure – size, dependency and communication frequency. Then the scheduler will allocate sufficient hardware resources to execute those tasks guided by a cost function in a greedy manner.

Experiments on a top supercomputer have shown great benefit from the

smart HPC scheduler when targeting large-scale workloads [67, 113]. It outperforms other state-of-the-art methods on both performance and scalability.

1.4.3 AccCOLAB: A Collaborative Scheduler using topological model for Multi-FPGA Clusters

A novel topological ranking heuristic based on lattice and representation theory is designed to model the heterogeneity of FPGA-based multi-accelerator clusters.

The applied representation is general enough to be used across different types of SoC topology (linear, ring, crossbar and etc.) for Multi-FPGA clusters, and is capable of representing the required resource and connectivity requirements for irregular workloads.

Equipped with the resulting heuristic, the runtime scheduler provides the collaborative ranking to order tasks on Multi-FPGA clusters with incomparable resource need and unpredictable execution time efficiently and then assigns sufficient re-configurable hardware resource in a dynamic manner.

Following this design, the topological ranking-based scheduler [110] has demonstrated a significant advantage over both the commonly used commercial FIFO scheduler, and other heuristic-based research schedulers on real world applications.

1.5 Publications:

Some of the material used in this thesis has been published in the following papers:

- [**CGO 20**] **T.Yu**, P.Petoumenos, V.Janjic, H.Leather, J.D. Thomson, COLAB: A Collaborative Multi-factor Scheduler for Asymmetric Multicore Processors. in *Proceedings of the International Symposium on Code Generation and Optimization*. ACM, 2020.

- [**TPDS 19**] **T.Yu**, W.Zhao, P.Liu, V.Janjic, X.Yan, S.Wang, H.Fu, G.Yang, J.D. Thomson. Large-scale automatic k-means clustering for heterogeneous many-core supercomputers. to appear in *IEEE Transactions on Parallel and Distributed Systems*. IEEE Press, 2019.

- [**PACT 19**] **T.Yu**, P.Petoumenos, V.Janjic, M.Zhu, H.Leather, J.D. Thomson, POSTER: A Collaborative Multi-factor Scheduler for Asymmetric Multicore Processors. in *Proceedings of the 28th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Press, 2019.

- [**SC 18**] L.Li, **T.Yu**, W.Zhao, H.Fu, C.Wang, L.Tan, G.Yang and J.D.Thomson. Large-scale hierarchical k-means for heterogeneous many-core supercomputers. in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*. IEEE Press, 2018.

- [**FPT 18**] **T.Yu**, B.Feng, M.Stillwell, L.Guo, Y.Ma and J.D.Thomson. Lattice-based scheduling for multi-FPGA systems. in *Proceedings of the International Conference on Field-Programmable Technology 2018*. IEEE Press, 2018.

1.6 Thesis Outline

The remainder of the thesis is organized as follows:

Chapter 2 presents all technical background of this thesis.

Chapter 3 discusses the related work and state-of-the-art targeting different heterogeneous architectures.

Chapter 4 shows the work on scheduler targeting asymmetric multi-core processors.

Chapter 5 shows the work on data partitioning and scheduler targeting hierarchical many-core supercomputers.

Chapter 6 shows the work on scheduler targeting FPGA-based multi-accelerator clusters.

Chapter 7 concludes the thesis and presents the future work.

Chapter 2

Technical Background

This chapter presents the technical background. It first describes the hardware background in the corresponding order to how the experimental chapters use these hardware platforms. It then presents the supporting system software, upon which the proposed software framework is built on. Finally, it describes the evaluation related issues – including metrics, benchmarks and real applications, which are applied to test the framework.

2.1 Heterogeneous Hardware

Heterogeneous architectures have been widely applied for modern computing systems, from distributed data centres and high performance supercomputers down to embedded multi-core processors. This section discusses three types of heterogeneous architectures used in this thesis.

2.1.1 Asymmetric multi-core processor: ARM big.LITTLE

Energy and power constraints are central in designing processors targeting small mobile devices, such as smartphones and IoT sensors. Single instruction set architecture (ISA) asymmetric multicore processors (AMP) are necessary for extracting a good energy and performance balance, and becoming increasingly popular in the era of limited power budget CPU design and dark silicon. AMPs introduce new opportunities and challenges, consisting of high performance out-of-order big cores and energy saving in-order little cores on the same chip.

This thesis targets the most prevalent commercial architecture, ARM big.LITTLE, to demonstrate the performance on asymmetric multi-core processors schedulers. ARM big.LITTLE has been widely applied in major smartphones chip providers, including Samsung Exynos, Qualcomm Snapdragon and Huawei HiSilicon.

This thesis targets a typical big.LITTLE architecture with 4 big and 4 little cores in ARM Cortex-A series processor. The big cores are out-of-order 2 GHz Cortex-A57 cores, with a 48 KB L1 instruction cache, a 32 KB L1 data cache, and a 2 MB L2 cache. The little cores are in-order 1.2GHz Cortex-A53, with a 32 KB L1 instruction cache, a 32KB L1 data cache, and a 512 KB L2 cache.

2.1.2 Hierarchical many-core supercomputer: Sunway Taihulight

More sophisticated than standard AMPs, many-core processors in modern supercomputers have been designed hierarchically. High performance multi-core processors used in supercomputers are usually grouped together to make core groups on a chip, and then multiple chips grouped into nodes with shared memory. Hierarchical multi-core processors in supercomputers work together on specific large-scale workflows in an intelligent way to achieve high system performance, measured by peta floating-point operations per second (PFLOPS). The recent announced top machine, Summit in Oak Ridge National Laboratory, equipped with IBM Power9 processors and Nvidia Tesla V100 GPGPUs has achieved 143.5 PFLOPS on LINPACK benchmarks.

This work targets one of the world's leading supercomputers (the top machine in the Top500 list during this project) – Sunway Taihulight supercomputer, equipped with SW26010 many-core processor – to demonstrate the proposed large-scale workflow oriented scheduler design. The basic architecture of SW26010 is shown in Figure 2.1. Each processor contains four *core groups (CGs)*, which are connected by network on chip (NoC). There are 65 cores in each CG, 64 *computing processing elements (CPEs)* which are organized in a 8 by 8 mesh and a *managing processing element (MPE)*. The MPE and CPE are both complete 64-bit RISC cores, but are assigned different tasks while computing. The MPE is designed for management, task schedule, and data communications. The CPE is assigned to maximize the aggregated computing throughput while minimising the complexity of the

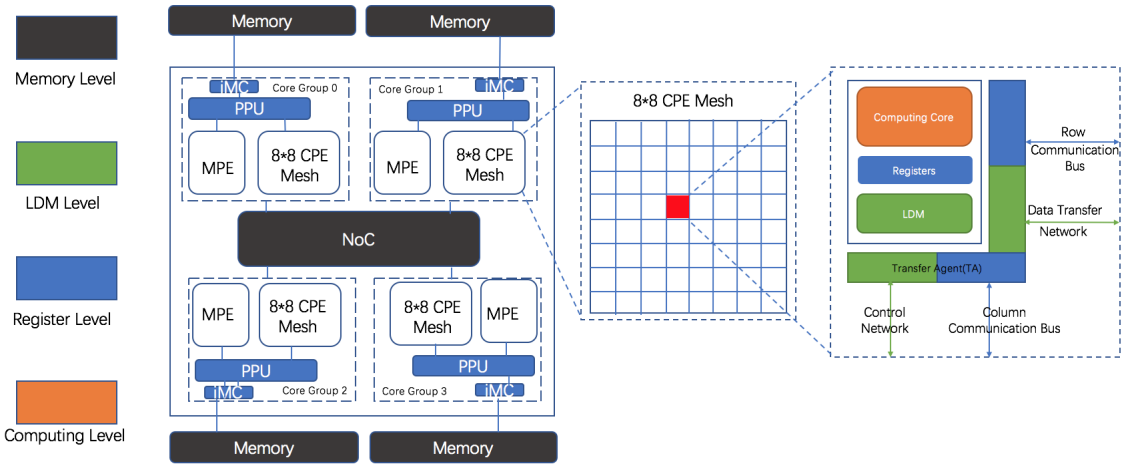


Figure 2.1: The general architecture of the SW26010 many-core processor

micro-architecture.

The SW26010 design differs significantly from the other multi-core and many-core processors: (i) for the memory hierarchy, while the MPE applies a traditional cache hierarchy (32-KB L1 instruction cache, 32-KB L1 data cache, and a 256-KB L2 cache for both instruction and data), each CPE only supplies a 16-KB L1 instruction cache, and depends on a 64 KB *Local directive Memory (LDM)* (also known as *Scratch Pad Memory (SPM)*) as a user-controlled fast buffer. The user-controlled 'cache' leads to some increased programming difficulty in for using fast buffer efficiently, and at the same time, providing the opportunity to implement a defined buffering scheme which is beneficial whole system performance in certain cases. (ii) Concerning the internal information of each CPE mesh, it has a control network, a data transfer network (connecting the CPEs to the memory interface), 8 column communication buses, and 8 row communication buses. The 8 column and row communication buses provide possibility for fast register communication channels to across the 8 by 8 CPE mesh, so users can attain a significant data sharing capability at the CPE level.

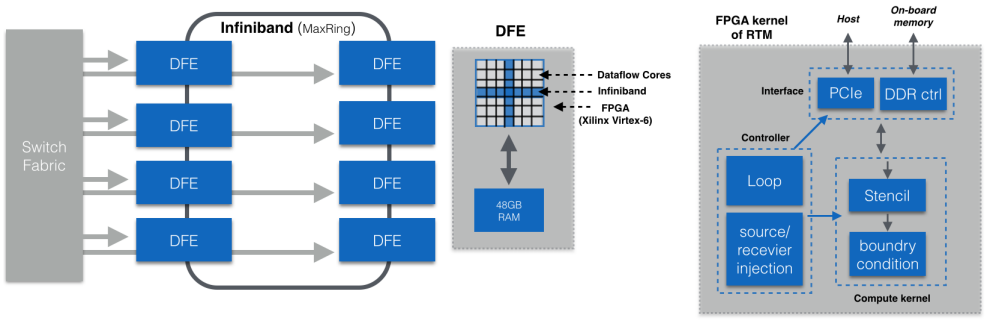


Figure 2.2: FPGA-based multi-accelerator system (MPC-X2000) and RTM kernel design

2.1.3 FPGA-based multi-accelerator systems for data centres: Maxeler MPC-X

Beyond general-purpose processors, accelerators like Field Programmable Gate Array (FPGA) have been shown to perform exceptionally well in terms of parallel performance and energy efficiency in modern commercial data centres and next generation cloud platforms [118]. For example, Microsoft claims that all its Azure cloud servers are now equipped with Altera Stratix FPGA [52] and Amazon AWS has begun to provide service from its EC2 server with FPGAs [79]. Intel acquired one of the leading FPGA technology providers, Altera for $167 billion in 2015 and is working on the next generation SoC with heterogeneous CPU+FPGA architectures. Intel predicts that more than 30% of cloud servers in the world will be equipped with FPGAs in 2020.

A typical FPGA-based multi-accelerator platform, MPC-X 2000 by Maxeler Technologies [7] is used to demonstrate the proposed FPGA-oriented scheduling work. This commercial device has been widely applied in oil & gas industry and financial analysis [29]. The MPC-X 2000 architecture is shown in the left-hand side of Fig. 2.2. It is comprised of eight DataFlow Engines (DFE) in which each of the DFEs is physically interconnected via a ring topology. All DFEs are the same specialized computation resources, each employing a Xilinx Virtex-6 SX475T FPGA to support reconfigurable designs and 48GB (or more) RAM for bulk storage. Under this setup, programs can dispatch computationally intensive tasks to single or multiple DFEs across

an Infiniband network. Each DFE cannot be separated to multiple applications simultaneously which means each DFE can only be assigned to single task during runtime scheduling.

2.2 System Software

System software has the duty to manage hardware resources and execute application programs. The proposed heterogeneity-aware scheduling and data partitioning framework is designed work across multiple systems levels, including the operating system scheduler and the middleware-level resource manager. This section presents the the system software background together with the GEM5 simulator, which is used across experiments.

2.2.1 Linux Kernel Scheduler

Resource management and multi-task scheduling are core functions for modern operating systems (OS). Most scientific high performance computing systems and commercial mobile devices use Linux-based OS, including the Red Hat Enterprise Linux (RHEL) for the Summit supercomputer and Google Android for smartphones.

The default Linux CFS scheduler was implemented by Ingo Molnar and merged in Linux v2.6.23 to replace the previous Linux vanilla scheduler. All scheduling related functions in Linux Kernel are grouped in the linux/ kernel /sched. The main scheduling interfaces with other OS components are listed and implemented in sched.h and core.c, and the default scheduling algorithm - Completely Fairness Scheduler (CFS) is implemented in fair.c

It is designed to model and support efficient multi-tasking co-execution on CPUs. It uses virtual runtime (actual runtime normalised to the total number of co-executed tasks) to track the execution of each thread without a pre-defined time slice and orders the runqueue (timeline of tasks) using a red-black tree data structure. Under this structure, the scheduler can pick the next task with smallest virtual runtime in the leaf nodes without further searching and comparing operations to achieve an O(logn) time complexity. Since the length of each CPU time slice are limited by the smallest virtual runtime of co-executed tasks, CPU time will then be split up as fairly as possible on the multi-tasking.

2.2.2 Data Centre Resource Manager

Distributed cloud data centres and high performance supercomputers often have additional customised resource managers as middleware which target large-scale workflow and scientific applications, such as web services and geographical simulations.

For example, Facebook designed and implemented Bistro [44], as a high-performance flexible distributed scheduling framework for its data centres and Google has its scalable scheduler, named Omega [88] for large-scale computing clusters. Maxeler implemented its Maxeler Orchestrator [73] scheduler on the Xilinx FPGA-based devices. It works in a round-robin manner to ensure fairness. The Sunway Supercomputer equipped Linux-based OS requires programmer assistance to schedule specific programs on suitable hardware resources.

2.2.3 Simulation

Simulators provide the flexibility to test a variety of configurations and hardware setups. Some of the work presented in this thesis is evaluated on the well-known full system simulator – GEM5.

Gem5 simulator [14] is an open-source cycle-accurate simulator built by many academic and industrial institutions, including AMD, ARM, HP, MIPS, Princeton, MIT, and the Universities of Michigan, Texas, and Wisconsin. It is the product of a project merge between the M5 and GEMS projects, and uses a liberal BSD-like licence. It combines high levels of configuration, including multi-ISA and CPU model support from M5 (ARM, ALPHA, and x86), and a detailed flexible memory system including multiple cache coherence protocols and interconnect models from GEMs. Gem5 provides a full system mode (FS mode) to simulate a complete system with devices and an operating system and the flexibility to run experiments from check points. Its detailed CPU mode provides sufficient performance counters, working similar with the Performance Monitor Units (PMU) in real hardware, to track runtime performance. The main drawback of GEM5 is that it that the simulation time is significantly larger than running on real hardware. It requires around one hour to simulate one second real execution, which limits the test sizes and the number of evaluations possible.

Since original released, GEM5 has been downloaded tens of thousands of times and is widely used in computer architecture research.

2.3 Performance Evaluation

This section presents the metrics and benchmarks/applications applied to test and evaluate the system performance.

2.3.1 Metrics

Heterogeneous Average Normalized Turnaround Time (H_ANTT)

A commonly used system-level performance metric is *Average Normalized Turnaround Time* (ANTT), as introduced in [39]. This user-oriented performance metric uses as the baseline execution time of each application when executed on its own, i.e. when there is no resource sharing and the execution is never interrupted or preempted, so scheduling decisions have little effect. It is the average slowdown of all applications in the mix relative to their isolated baseline runtime. It is defined as follows:

$$ANTT = \frac{1}{n}\sum_{i=1}^{n}\frac{T_i^M}{T_i^S}$$

Where T_i^M denotes the turnaround time of program i under the multi-program mode and depends on its co-executing programs. The T_i^S denotes the turnaround time of program i under the single-program mode. While this original metric fails to work as intended for AMPs. The baseline execution time T_i^S when executed alone is still affected by scheduling decisions if processor cores are difference. For example, under the default Linux fairness scheduling policy on AMPs, the first application thread might be allocated to either a high-frequency out-of-order big core or a low-frequency in-order little core, which will leads to different future scheduling space and then results in different performance. So T_i^S fails to work as a stable baseline.

In this thesis, a modified metric *Heterogeneous Average Normalized Turnaround Time* (H_ANTT) is designed to quantify scheduling efficiency on AMPs and overcome the above problem. It updates the baseline performance by using the execution time of each application in single-program mode on a corresponding symmetric configuration - *where all configured processor cores are big cores*. T_i^{SB} is used to denote this updated baseline performance of program i in single-program mode with full big cores configuration. The modified metric is defined as follows:

$$H\_ANTT = \frac{1}{n}\sum_{i=1}^{n}\frac{T_i^M}{T_i^{SB}}$$

When evaluating on a single benchmark on its own, this work uses the *Heterogeneous Normalized Turnaround Time* (H_NTT):

$$H\_NTT = \frac{T^M}{T^{SB}}$$

H_ANTT and H_NTT are better when lower.

Heterogeneous System Throughput (H_STP)

Another well-know system-level performance metric concerned is *System Throughput* (STP). This system-oriented metric is also described in [39]. STP is the sum of the throughputs of all co-executed applications in the multi-program mode, relative to their isolated throughput in the single-program mode as following:

$$STP = \sum_{i=1}^{n}\frac{T_i^S}{T_i^M}$$

Similar with the issue on ANTT, modification is needed before applying STP on asymmetric processors because the baseline performance from solo execution is affected by the scheduler. Thus, a modified metric Heterogeneous System Throughput (H_STP) is proposed. It uses the same updated baseline case as H_ANTT. Applying the terms above, H_STP is defined as:

$$H\_STP = \sum_{i=1}^{n}\frac{T_i^{SB}}{T_i^M}$$

H_STP is better when higher.

Workload Completion Time (WCT)

The core metric of performance targeting large-scale workload oriented HPC and customized FPGA-based multi-accelerator system is the workload completion time WCT, also known as makespan [62] for each given multi-workload

scenario. Note that the mapping $\mathcal{P}$ is determined by the given workloads scenario S and the applied approach $\digamma$, then there is:

$$WCT(\mathcal{P}) = WCT(S, \digamma) = max.\ wTime_k(\digamma), W_k \in S$$

WCT is the time when the final co-executed workload is finished by the system including scheduling and system overhead [62].

One Iteration Completion Time (OICT)

When evaluating the system performance targeting a single non-deterministic algorithm based workload, a specialized workload completion time, the one iteration completion time (OICT) is used. This is because the number of iterations needed for the program to complete is related to the initial distribution of randomised start points. The total workload completion time cannot directly represent the real system-level performance. OICT is widely applied in the research community to avoid the performance influence from random initialization issues [83].

Percentage of Upper-Bound

This metric shows the percentage between the solution of the tested approaches and the theoretical performance upper-bound. It is designed to normalize the original results and then easily present them:

$$PUB(S, \digamma) = WCT_{upper}(S)/WCT(S, \digamma)$$

2.3.2 Benchmarks

This thesis tests some well-known benchmarks including PARSEC3.0 [13], SPLASH-2 [106], k-Means [67] and F11 [28] to evaluate the system performance of targeted heterogeneous systems.

PARSEC3.0

PARSEC3.0 is a multi-threaded benchmark suits developed by researchers from Princeton. PARSEC is widely applied by the research community and has many plus points, including inclusion of modern, emerging workloads and the diversity of underlying applications with both computing-intensive

and memory-intensive programs. It provides parallel editions of benchmarks using both pthreads and OpenMP and the flexibility on the size of dataset from *simsmall, simlarge to native*. A representative selection of subsets is applied which can run within a reasonable amount of time on Gem5 simulated ARM architecture, including: blackscholes, bodytrack, dedup, ferret, freqmine, swaptions and fluidanmate.

SPLASH-2

SPLASH-2 is a multi-threaded benchmark set mainly targeting HPC applications, which has been integrated into PARSEC3.0 environment. A selection of benchmarks from this set is used to enrich the test set.

K-Means

k-means is a well-known clustering benchmark, used widely in many AI and data mining applications, such as bio-informatics, image segmentation, information retrieval and remote sensing image analysis. This thesis uses parallel k-Means algorithm as the benchmark to evaluate it supercomputer-oriented scheduling and data partitioning framework. The input dataset of K-means comes from both UCI Machine Learning Repository and ImageNet Large Scale Visual Recognition Challenge (ILSVRC2012) [84] to test the system performance on different scales.

UCI Machine Learning Repository UCI Machine Learning Repository is a collection of datasets and benchmarks that are widely used by the machine learning community for the empirical analysis of algorithms. A representative selection from this repository is used including Kegg Network, Road Network and US Census 1990. Characteristic of these benchmarks are briefly presented in the upper half of Table 2.1, where n is the number of data samples, k is the number of targeting clusters and d is the dimension of data samples.

ILSVRC2012 The ImageNet Large Scale Visual Recognition Challenge (ILSVRC2012) [84] is a dataset in the field of large-scale image classification and object detection. Originally released in 2010, it runs annually and has attracted world-wide participation. A high-dimensional dataset from this source is tested to demonstrate efficiency of the proposed supercomputing-oriented large-scale data partitioning framework, which briefly presents in the lower half of table 2.1.

Table 2.1: Benchmarks from UCI and ImgNet

Data Set	n	k	d
Kegg Network	6.5E4	256	28
Road Network	4.3E5	10,000	4
US Census 1990	2.5E6	10,000	68
ILSVRC2012	1.3E6	160,000	196,608

F11 Benchmark

Genetic Algorithms (GAs) are commonly used in the field of optimisation research to find good solutions by efficiently using time when targeting large-scale data scenarios. Additionally, parallel GAs implemented in multi-FPGA systems usually result in higher quality solutions due to its parallel feature to search on the solution space. Function 11 (F11) is a well-known benchmark [28] to test the performance of GA, which aims to find the optimisation solution to maximise the following:

$$f(x) = 1 + \sum_{n=1}^{N} x_n^2/4000 - \prod_{n=1}^{N} cos(X_n)$$

Where the domain of x_n and value of N define the size of the input and the solution space.The parallel GA kernel design is similar with the RTM kernel but the computation component is SCM (select, crossover and mutation).

The inputs of parallel GA have different sizes including small, medium and large. A mixed domain, named *mix-L*, is considered to increase the complexity and heterogeneity of the input with corresponding solution space, where the object is to find the optimal solution in two different domains simultaneously. In the experimental setup, the user can submit multiple GA workloads. For each workload, the user asks to find the optimization solutions of F11 in different input scenarios, which contains different input sizes and irregularity levels.

2.3.3 Real Applications

An representative selection of scientific applications are used to validate the high performance of the proposed approach on real systems, which includes the Reverse Time Migration (RTM) from oil and gas industry, Gene Expression Classification, VHR Remote Sensing and Land Cover Classification.

Reverse Time Migration

Reverse Time Migration (RTM) is a practical method commonly used in the Oil and Gas industry to model the bottom of salt bodies in the Earth's subsurface. It is based on the computation of the isotropic acoustic wave equation through two propagations: a forward propagation which takes input velocity model and initial traces to compute the forward wavefield and a backward propagation which uses the same velocity model and the forward wavefield to compute the targeting image of subsurface. This process is named as a *shot* or RTM kernel. 3D stencil-based computation is usually applied to process each shot. Normally 10^5 to 10^7 number of shots [2] should be computed simultaneously to analyse certain area of the subsurface due to the different size of area which definitely leads to a computation-intensive workload. It is an ideal HPC application which has been widely used to test the performance of accelerator-based heterogeneous systems [42, 29, 2].

This original trace is based on the Sandia/SEG Salt Model 45 shot subset which is open source.[1] A high level dataflow-oriented RTM design is shown in the right-hand side of Fig. 2.2. The RTM kernel design on FPGA is applied from [42].

The user can submit multiple RTM workloads which contain irregular RTM tasks (shots) in different sizes. One shot is referred to one task in this thesis to make the presentation consistent.

Gene Expression Classification

Genomic information from gene expression data has been widely used on improving clinical decision and molecular profiling based patient stratification. Clustering methods, as well as their corresponding HPC-based solutions[103], are adopted to classify the high-dimensional gene expression sequences into known patterns, which indicates that the number of targeted clustering centroids are determined in advance. As it is well known that there are still large numbers of gene expression sequences among which the patterns are not yet discovered. Therefore, the proposed auto-clustering method can potentially help find new patterns from high-dimensional gene expression datasets.

In this work, the proposed supercomputing-oriented approach is tested on the ONCOLOGY & Leukaemia gene expression datasets[41]. There are 4254 subjects and each subject has 54675 probesets, which means the dataset

[1] http://wiki.seg.org/wiki/SEG_C3_45_shot.

contains 4254 data elements in total and each data element has 54675 dimensions.

Land Cover Classification

This is a popular remote sensing problem, requiring unsupervised methods to handle high numbers of unlabelled remote sensing images [70]. In recent years, high-resolution remote sensing images have become more common in land cover classification problems. The problem definition on high-resolution images is more complex as the classification sample can be a block of pixels instead of one pixel, which means the dimensions of high-resolution data samples can be even larger than normal remote sensing images.

This thesis tests on a public dataset called Deep Globe 2018 [33], and classifies the images into seven classes, representing the urban, the agriculture, the rangeland, the forest, the water, the barren and unknown. There are 803 images in the Deep Globe 2018 dataset, and each image has about $2k \times 2k$ pixels. The resolution of the image is 50cm/pixel. In this problem definition, one image needs to be clustered, where n is 5838480, k is 7 and d is 4096.

Chapter 3

Related Work

This chapter presents the related work and state-of-the-art research for this thesis. It first presents the work related to heterogeneity-aware schedulers targeting asymmetric multicore processors, and then targeting hierarchical many-core processors with data partitioning, and finally targeting FPGA-based multi-accelerator systems. In each section, it first describes the general issues and developments of the related work with a qualitative analysis, and then presents the most representative state-of-the-art work on each issues in detail in subsections. The research in this thesis is built on these foundations.

3.1 Schedulers for asymmetric multicore processors

Efficient, heterogeneity-aware scheduling for asymmetric multicore processors has attracted attention during the recent decade.

The first issue concerned in the literature is how to accelerate bottlenecks in the multi-threaded programs using the high performance big cores. A bottleneck can be defined as any code segment which blocks the execution of other threads during runtime. Dealing with this issue is critical for performance of multicore systems on parallel executions – for instance, a barrier where multiple threads need to reach a synchronisation point before others can make the next progress. Other bottlenecks can be critical sections of multi-thread programs and pipeline stages. Failure to accelerate bottlenecks can lead to reduction or even cessation of parallel performance scaling, which is usually expected from parallel execution on multicores. The first effort was

made by Suleman et al [96], which can efficiently detect and accelerate one bottleneck at a time on single big core systems. The state-of-the-art bottleneck detection and acceleration technique was presented by Joao et al [60]. The BIS approach designed in this paper can efficiently and accurately identify which bottlenecks are most critical at any given time and accelerate them accordingly on big cores.

The second issue is about fairness. Fairness, or requiring that all threads make similar progress, is a critical goal for scheduling on heterogeneous multicores, especially when targeting multi-threaded workloads with barrier-synchronisation and multi-program workloads with quality-of-service (QoS) constraints. The default Linux CFS scheduler follows a complete fairness way on homogeneous processors. Further, researchers have began to consider achieving fairness on heterogeneous multicore processors. Craeynet et al. made the first effort on using a performance impact estimation (PIE) model [100] to predict the runtime behaviours and then they designed the most representative work [99] named as equal-progress scheduling. Kim, et al [64] proposed a more recent fairness guaranteed scheduler targeting asymmetric cores, but their formulations only suited for single-threaded program with deterministic load of each thread.

Followed by the two concerns above, researchers have began to consider the core sensitivity between different threads and tried to map the most suited threads on big cores by their predicted speedup. Cao et al. initially proposed a speedup-aware scheduler named *yin* and *yang* [21] which binds threads on asymmetric cores based on static classification. ARM [56] designed its GTS scheduler by intelligent runtime migration between asymmetric cores based on threads runtime behaviours. Further, Jibaja et al. proposed the state-of-the-art WASH algorithm [58] by dynamically and periodically predicating threads speedup using machine learning based models with performance counters support.

Subsequently, research schedulers have been designed to address all above issues simultaneously to result in better performance on variant environments. The table 3.1 presents a qualitative analysis on related work, where COLAB is the proposed approach in this thesis. In this section, the most representative works for the three issues - bottleneck acceleration [60], fairness [99] and core sensitivity [56, 58] are described in detail below. The proposed COLAB scheduler in this thesis is motivated by the techniques from all these approaches. The WASH scheduler by Jibaja et al [58] represents the most state-of-the-art approach which is compared against the COLAB

in the corresponding experiment chapter.

Table 3.1: Qualitative Analysis on Heterogeneity-Aware Schedulers Targeting Asymmetric Multicore Processors

Approaches	Core Sens.	Fairness	Bottle-neck	Collaborative
Suleman, et al. [96]			✓	
Craeynest, et al. [100]	✓	✓		
Craeynest, et al. [99]	✓	✓		
Cao, et al. [21]	✓			
Joao, et al [60]	✓		✓	
ARM [56]	✓		✓	
Kim, et al [64]	✓	✓		
Jibaja, et al [58]	✓	✓	✓	
COLAB	✓	✓	✓	✓

3.1.1 J. Joao et al. (2012)

The state-of-the-art Bottleneck Identification and Scheduling (BIS) approach on asymmetric multi-processor was presented by Joao et al. in [60]. The key idea is to measure the number of cycles spent by each thread when waiting for bottlenecks and then rank and accelerate those bottlenecks based on those values. There are two main elements:

1. Bottleneck Identification: A bottleneck table (BA) is implemented in which each entry corresponds to a bottleneck and includes a thread waiting cycles field (WCT) to track this information. WCT value is computed by aggregating the number of threads which are waiting on this bottleneck. The critical bottlenecks are then identified by BA with N highest WCT values where N is based on the number of big cores in the system.

2. Bottleneck Acceleration: An acceleration index table (AIT) is created associated with each small core to decide whether a bottleneck should be accelerated by big cores. The small cores send bottleneck execution requests based on the information from AIT and those requests are then enqueued in a scheduling buffer (SB) in big cores. SB is actually a priority queue which run the oldest instance with highest WCT value. The SB also implements an request-abort function to avoid false serialisation and resource starvation, which will let the big core send back a bottleneck to small cores if this bottleneck does not have the highest WCT value and is ready to run on small cores.

3.1.2 K. Craeynest et al. (2013)

Two fairness-aware scheduling methods are presented by Craeynet et al. in [99].

1. The first is called *equal-time* scheduling. It trivially keeps each thread running on each type of core in the same amount of time slices, and is implemented by round-robin or random selection of the thread which currently is on the small core to schedule on big core next. However, it cannot guarantee true fairness if threads experience different slowdown or speedup from different cores.

2. A more advanced approach to handle this issue considered in their paper is called *equal-progress* scheduling, which guarantees each thread achieves equal progress on heterogeneous cores. The key challenge of this approach is to estimate a big-versus-small-core scaling factor for all threads, and this can be used to compute the slowdown between running on heterogeneous cores and big cores in isolation. There are three methods provided to obtain this factor: (1) sampling-based method which compute this factor during a sample phase as the ratio between CPI (clock cycles per instruction) on different cores and then apply on symbiosis phase; (2) history-based method which is similar with the sampling way but records historical CPI values and continuously adjust the ratio; (3) model-based method which use a performance impact estimation (PIE) [100] analytical model to estimate this factor with hardware support.

Also presented is a *guaranteed-fairness* approach, designed for when system throughput needs to be of concern. A threshold of fairness can be setup and the scheduler will defer to a throughput-optimisation policy after this threshold has been reached.

Overall, the fairness-aware scheduler achieves a 14% performance gain on homogeneous multi-thread workloads and a 32% performance gain on heterogeneous multi-thread workloads on average against other schedulers.

3.1.3 ARM (2013)

ARM has designed a Global Task Scheduling (GTS) technique [56] through the development of asymmetric multicore processors, called big.LITTLE Multiprocessing (MP). In this technique, the differences in compute capacity between big and little cores are given to the OS. The OS scheduler tracks the performance requirement for all software threads, including both application threads and the system calls, and then uses that information to decide whether the big or the little core to use for each.

When implementing the ARM big.LITTLE MP, the OS scheduler needs to decide when a software thread can run on a big core or a little core. This is achieved by comparing the tracked load of software threads against predefined load thresholds, which include an up migration threshold and a down migration threshold. In more detail, a little core thread will be considered to migrate to a big core if the tracked load average of this thread exceeds the up migration threshold. Similarly, a big core thread will be considered to migrate to a little core if the load average of it drops below the down migration threshold. To keep fairness across cores in both big and little clusters, the standard Linux CFS scheduler applies within clusters. There are five migration mechanisms to determine when to migrate a task between different type of cores:

1. Fork migration: When the fork system call is used to create a new software thread, this mechanism is invoked to migrate the new thread to a big core.

2. Wake migration: When a previously idle thread becomes ready to run, this mechanism is invoked to decide which type of cores executes the thread. ARM big.LITTLE MP uses the tracked information and the general assumption is that this thread resumes on the same cluster as before.

3. Forced migration: When a long running thread on a little core do not sleep or do not sleep very often and the tracked load of it exceeds the up migration threshold, this mechanism will be applied to migrate this thread to a big core.

4. Idle pull migration: When a big core has no thread to run, this mechanism will be invoked to check all little cores and then migrate the most suited running thread to this big core.

5. Offload migration: This mechanism works to periodically migrate big core threads downwards to little cores to make use of unused compute capacity if the Linux CFS load balancing is disabled.

The main advantages of this technique can be summarised as follows: it supports flexible hardware configurations; the targeting architecture can have different numbers of different types of cores and any number of cores can be active at any time to increase peak performance; it supports flexible multiple threads co-execution between heterogeneous cores; computing-intensive application threads can be isolated on the big core cluster while light-weight system calls are bound on the little core cluster. This enables application tasks to complete faster without disturbed by additional background system calls. Further, interrupts can be targeted individually either to big or little cores.

3.1.4 I. Jibaja et al. (2016)

WASH presented by Jibaja et al. in [58] is the state-of-the-art scheduler which provides an Asymmetric Multicore Processors (AMP) aware runtime environment and which uses dynamic analysis to classify applications, identify bottlenecks and prioritise threads based on single-ISA. It is the first work which can optimise critical path, core sensitivity, priorities and load balancing simultaneously. This scheduler starts with scheduling application threads on big cores and VM threads on small cores, then dynamically classifies (scalable or not), and prioritises and migrates threads guided by the runtime information.

They implement two models to record and provide this information.

1. *Dynamic bottleneck analysis.* The core function is to accelerate threads which hold contended locks by computing a ratio between the time this

thread waits for another thread to release a lock and the total execution time so far. It selects the one which requires others wait the longest to be executed on big cores sets priority for these threads.

2. *Core sensitive predictor.* This method uses linear regression and Principal Component Analysis to learn most significant performance counters and the corresponding weights. The selected counters include such as INSTRUCTIONS_RETIRED and L1D_ALL_REF:ANY for Intel processors and RETIRED_UOPS and CPU_CLK_UNHALTED for AMD cores. Those counters can then be used to compute the speedup if the scheduler were to assign a thread to a big core rather than a small core. It uses two relative ranking functions to achieve this. The *ExecRank* is an adjusted priority which shows how much performance gain can be achieved if the thread is on big cores by accumulating on retired instructions. The *LockRank* is to show the bottleneck level based on the amount of time other threads have been waiting for it.

Overall, WASH reports a 20% performance gain and a 9% energy saving against prior work on heterogeneous multicore AMD cores in different configurations. This thesis views WASH as the overall state-of-the-art heterogeneity-aware scheduler on asymmetric multicore processors,

3.2 Schedulers and data partitioning for hierarchical many-core supercomputers

Scheduling on general-purpose processors and HPC requires different scheduling decisions. The design of schedulers targeting hierarchical many-core supercomputers needs to address additional heterogeneities. These additional heterogeneities mainly result from the large-scale distributed processors, the hierarchical memory architectures and specialised hardware resources integrated into the supercomputers. The resulting runtime issues need to be tackled, including the distributed load imbalance and the specialised resource management.

Load Balancing: Large-scale parallel algorithms running HPC applications usually follow a Single Instruction Multiple Data (SIMD) model, which means parallel threads executing the same code segment on different dataflow.

However, unfairness between parallel threads can still happen on such data-parallel pattern when there are imbalanced input data distributions, asymmetric network topology and differing task execution times before reaching the syncretisation point. Multiple efforts have been made to balance the load of a supercomputer during runtime scheduling. Schloegel et al [87] proposed a scheduler to analyse the parallel dataflow and then compute the balanced data distribution. Duran et al [37] tried to automatic balance the load by assigning more processors to the parallel processes which need longer computing time. The most representative balancing scheduler is designed by Boneti et al [18] by automatic load re-balance.

Specialised Resources: State-of-the-art HPC schedulers have begun to consider specialised resources on supercomputers in additional to classical CPUs, named as multi-resource schedulers. Instead of previous CPU-centric HPC schedulers, multi-resource schedulers make decisions not only based on the computing/CPU requirements of the tasks, but also on the difference between storage and communication needs. These kind of decisions are much more suited to targeting massive data-intensive workloads in the coming Big Data era, where there are high demands on runtime storage. Yoo et al, present the Slurm [108] scheduler, which provides a simple approach by sequentially allocating waiting jobs to fill the CPU and buffer following a First-Come-First-Serve (FCFS) policy – this allocator suffers a low resource utilisation and will be interrupted when either the CPU or buffer is full. Wallace et al. [102] design a dynamic power-aware approach to handle the multi-resource scheduling more efficiently. It applies a window-based scheduling policy to sort the data and then used a constraint model with runtime power estimation to achieve the multi-resource optimisation. Hung et al [53] consider the multi-resource scheduling problem as linear programming (LP) model with heterogeneous-aware conditions. Their Tetrium scheduler equipped with joint heterogeneous-aware task placement and job scheduling policy, represents the multi-resource scheduling as a single-objective optimisation problem under their LP model and provides a collaborative solution. The most current multi-resource scheduler is designed by Fan et al [40] using a multi-objective optimisation (MOO) model.

The table 3.2 presents a qualitative analysis on related work, where the proposed SupCOLAB scheduler in this thesis is listed at the end. The SupCOLAB scheduler is the first supercomputer-oriented scheduler addressing both load balance and specialised resources, targetting multi-objectives. In the remainder of this section, the most relevant work for the two supercomputer-

Table 3.2: Qualitative Analysis on Heterogeneity-Aware Schedulers Targeting Hierarchical Manycore Supercomputers

Approaches	Large-scale Load Balance	Specialized Resources	Multi-Objective Optimization
Schloegel et al [87]	✓		
Duran et al [37]	✓		
Boneti et al [18]	✓		
Yoo et al [108]		✓	
Wallace et al [102]		✓	
Hung et al [53]		✓	
Fan et al [40]		✓	✓
SupCOLAB	✓	✓	✓

specific issues - large-scale load balancing [18] and specialized resources [40] are described.

Data partitioning for hierarchical many-core supercomputers Beyond the schedulers, large-scale parallel algorithms running on specific supercomputers can be easily custom-designed and partitioned to fit given hardware. For example, the programmer can generate parallel threads corresponding to the number of assigned cores and partition the dataflow to fit the distributed storage. This thesis demonstrates the efficiency of the proposed SupCOLAB approach on supercomputers by a concrete study on k-means based large-scale workloads. This section presents the background work on heterogeneity-aware data partitioning methods to run k-means algorithms on modern supercomputers.

Kumar, et al [65] made the most relevant contribution in implementing the basic dataflow-partition based parallel *k-means* on the *Jaguar* (the previous version of *Titan*), a Cray XT5 supercomputer at Oak Ridge National Laboratory evaluated by real-world geographical datasets. Their implementation applies MPI protocols for broadcasting and reduction, and allows the scaling the value of k to more than 1,000 by using sufficient many-core processors in the supercomputer. Cai, et al [20] designed a similar parallel

Table 3.3: Qualitative Analysis on Heterogeneity-Aware K-Means Data Partitioning Targeting Hierarchical Manycore Supercomputers

Approaches	Samples n	Clusters k	Dimensions d
Kumar, et al [65]	10^{10}	1000	30
Rossbach, et al[83]	10^9	120	40
Bhimani, et al[10]	10^6	240	5
Cai, et al [20]	10^6	8	8
Ding, et al[35]	10^6	10,000	68
Bender, et al [9]	370	18	140,256
SupCOLAB	10^6	**160,000**	**196,608**

approach on *Gordon*, an Intel XEON E5 supercomputer at San Diego Supercomputer Center for grouping game players. They applied a parallel R function, *mclapply*, to achieve shared-memory parallelism and test different degree of parallelism by partitioning the original data-flow into different numbers of sets. They did not focus on testing the scalability of their approach but evaluated on the quality of the clustering.

Rossbach, et al[83] concentrate their efforts on implementing parallel k-means on heterogeneous architectures using GPU clusters to accelerate computing-intensive processes. They evaluated on 10 heterogeneous nodes – each node consisting of an NVIDIA Tesla K20M GPU with two Intel Xeon E5-2620 CPUs. The clustered a large-scale dataset with 10^9 data elements successfully. Bhimani, et al[10] further developed the GPU based data partitioning approach to cluster large datasets with more than 200 targeting centroids efficiently.

Ding, et al[35] reduce the unnecessary computation using triangle inequality. This design resulted in a significant advantage in performance and program scalability. This led to the first parallel k-means approach which can efficiently cluster the high-dimensional datasets with more than 10,000 dimensions for each data element on commodity Intel i7-3770K processors without additional support from accelerators. The most current approach on large-scale k-means is made by Bender, et al [9] through a 2-level hierarchical data partitioning, which can handle large-scale complicated dataset with more than 140,000 data dimensions.

The most relevant dataflow partitioning work [65] and the state-of-the-art effort on 2-level partitioning [9] are illustrated in subsections below.

3.2.1 C. Boneti et al. (2008)

Boneti et al. designed a static self-balancing resource allocator [17] and a dynamic scheduler [18] by letting the OS kernel automatically apply the specialised hardware priorities from IBM Power5 processor to the parallel processes without the programmer/user hints.

It is implemented as a new scheduling class in Linux kernel, named *HPCSched*, containing two scheduling policies (FIFO and round-robin), a load imbalance detector by learning from execution history with two heuristics (uniform and adaptive prioritization) and some architecture-dependent mechanism.

Experiments on different micro-benchmarks and a real application - the SIESTA simulation, demonstrate that their dynamic balancing scheduler does better than both previous static allocator and the default Linux kernel with more than 10% performance gain.

3.2.2 Y. Fan et al. (2019)

Fan et al [40] consider and model the multi-resource scheduling as a multi-objective optimization (MOO) problem and design a state-of-the-art multi-resource scheduler, named BBSched.

A representative specialised resource which BBSched can address efficiently is the burst buffer. It is designed to efficiently handle data-intensive workloads in modern supercomputer, which is an intermediate storage layer between processors and the main parallel file systems in shared memory. Composed by SSD, burst buffer can be either associated to single node or shared between nodes and enjoy a higher bandwidth and lower latency than main memory it is accessing.

Contrary to the previous single-objective model [53], BBSched equipped with genetic search algorithm in MOO space, provides an approach to optimize the utilization of multiple resources including both the processors and buffers, simultaneously and then return a Pareto set of solutions.

It gives the flexibility to the system managers to finally select the solution targeting site-specific metrics. BBSched also follows a window-based

scheduling policy, but it provide an upper-bound on number of iterations to prevent job starvation.

3.2.3 J. Kumar, et al. (2011)

The work of Kumar, et al [65] on designing a massively parallel k-means clustering algorithm targeting the *Jaguar* supercomputer is the most relevent to this thesis. Their work is motivated a use case of identification of geographic ecoregions. Without an efficient large-scale clustering method, this could only be done by human experts before, which are neither transparent nor repeatable.

Their work focuses on development of a clustering method for analysis of very large datasets using tens of thousands processors on a supercomputer. The parallel code in their implementation is written in the C language using the Message Passing Interface (MPI). The initial input file and seed centroids are read by the first process, and then are broadcast to all the processes. Every MPI process operates on a partial section of the initial dataflow, computes the distance calculation of points from the centroids and assigns it to the closest centroid. Each MPI process then calculates the partial sum along the points in each cluster for the next centroid calculation. At the end of each iteration, a global reduction operation, MPI_Allreduce is carried out, after each parallel process finishes its own calculation, to compute the new cluster centroids. Iterations are carried out until fewer than 0.05% data points change their cluster assignment from the previous iteration.

Two optimisation techniques are applied in their approach to accelerate the parallel computation:

- The first concerns triangle inequality. It established an upper limit on distances between data points and centroids to reduce the number of Euclidean distance calculations. Under this technique, unnecessary point-to-centroid distance calculations and comparisons can be eliminated based on the previous assignment and the new inter-centroid distances. In more detail, if the distance between a centroid assignment and a new candidate centroid is greater than or equal to the distance between this centroid and the next point, then calculation of distance between the next point and the new candidate centroid is not required.

- The second concerns empty clusters and centroid warping. A global

MPI reduction operation is carried out to identify a cluster with the highest average distance whenever there is an empty cluster. Once this cluster is chosen, the farthest point in that cluster is identified and is assigned to the empty cluster.

They use real-world datasets to evaluate their approach on the *Jaguar* supercomputer. By applying their massively parallel dataflow partitioning and the acceleration techniques, they can cluster a 84 GB dataset with 1,024,767,667 into 1000 centroids within 1000s.

3.2.4 M. Bender et al. (2015)

Bender, et al [9] investigated a novel parallel implementation proposed for *Trinity*, the latest National Nuclear Security Administration supercomputer with Intel Knight's Landing processors and their *scratchpad* two-level memory model. Their approach is the most current comparable work against the proposed SupCOLAB method which can not only partition dataflow, but also partition the number of target clusters k by their *hierarchical* two-level memory support – cache associated with each core and *scratchpad* for sharing.

Adapted originally from [45], their partitioning algorithm partitioned the input dataset into $\frac{nd}{M}$ sets, where M is the size of the *scratchpad*, n is the size of the dataflow and d is the number of dimensions of each data element, and then reduced $k\frac{nd}{M}$ centroids recursively if needed. Based on this partition, their approach scaled d into the 140,000s.

However each centroid, k, is a d-dimensional vector. The maximum value of $k * d$ is limited by the shared *scratchpad*. There are two main drawbacks to this approach:

- Firstly, only one of k or d can be scaled to a large number, as shown in Table 3.3. This fundamental bottleneck in their approach is because of the two-level memory. It is still impossible to partition and then scale all n, k and d independently through a 2 levels hierarchical architecture. This leads to the interaction constraint between k and d as discussed in their paper:

  $$Z < kd < M$$

  where Z is the size of cache. This partition-based method is not efficient if all k centroids can fit into one cache.

- Secondly, the performance scaling of 2-level data partitioning is shown to be poor as k or d grows towards the high end of possibility for this approach. Therefore even if the memory limits were somehow solved in some other way, the performance scaling would limit the growth of k or d.

These difficulties show the need for a new approach if larger values of k and d are to be reasonably handled. The three-level hierarchical SupCOLAB approach proposed in this thesis, addresses both issues of independent growth of k and d, and of scalability.

3.3 Schedulers for multi-accelerators systems

In addition to scheduling the special heterogeneities from hierarchical many-core supercomputers, more complex heterogeneities need to be considered when designing schedulers for FPGA-based multi-accelerator systems, including optimising data communication, configuration detection and addressing topology of FPGAs.

Based on the hardware parallelism from FPGAs, the first critical problem is how to optimise the schedule of runtime data communication cased by data dependence between parallel executions. J. Shu et al [95] and H. Topcuoglu et al. [97] proposed graph representations and modelling to address the communication issues.

The second additional heterogeneity is brought about by the reconfigurability of FPGAs. This results in a more difficult scheduling space as tasks can have different shapes (number of required and their topologies) and then are more suited to different hardware configurations. For example, based on the increasing size of input data, a code segment might get better performance by being allocated and running on more co-executed FPGAs. The scheduler should address this heterogeneity by determining the best configuration of a given tasks and then schedule it to sufficient resources. Multiple dynamic FPGA schedulers have considered this problem, which will be described in detail later.

The third issue concerning the multi-FPGA architecture is the topology of FPGAs. Given a multi-FPGA architecture, FPGAs might be directly connected or not. This asymmetry leads to a further complicated scheduling space for multi-FPGA tasks. For example, even if a task is allocated

to a deterministic number of FPGAs with same configuration, the computing ability of those FPGAs might differ. Unfortunately, no state-of-the-art FPGA scheduler has an efficient solution to address this problem. The Table 3.4 provides a qualitative analysis on the related FPGA schedulers and claims that only the proposed AccCOLAB scheduler can address all the heterogeneities efficiently.

Table 3.4: Qualitative Analysis on Heterogeneity-Aware Schedulers Targeting FPGA based Multi-Accelerator Systems

Approaches	Optimizing Communication	Configuration Detection	Multi_FPGA	Topology
Shu et al [95]	✓		✓	
Topcuoglu et al. [97]	✓		✓	
Handa et al [49]	✓	✓		
Redaelli et al [80]	✓	✓	✓	
Jing et al. [59]	✓	✓	✓	
AccCOLAB	✓	✓	✓	✓

3.3.1 J. Suh et al. (2000)

J. Shu et al [95] proposed a heuristic-based scheduler on multi-FPGA systems targeting a specialised task – optimising the runtime communication. Motivated by the I/O limitation, efficiently executing the communication task is critical for the overall system performance on multi-FPGA.

They apply a graph representation to model the tasks, in which each task is a node with weight value on computation time and each edge from two nodes denote the directed communication need. They design and implement a heuristic algorithm, based on breadth-first search on the tasks graph, and then assign tasks in a bottom-up fashion. This simple algorithm achieves an O((M+ N log N) time complexity to provide a high performance solution for the NP-hard scheduling problem, where M is the number of tasks and N is the number of edges – the number of communications.

As detailed in their reported results on a SUN workstation with 100 random workloads, their approach can achieve around 20% speedup (reducing latency) over others.

3.3.2 H. Topcuoglu et al. (2002)

A more sophisticated approach has been proposed to address data dependence issues by H. Topcuoglu et al.

The Heterogeneous Earliest Finish Time (HEFT) algorithm [97] is a list scheduling algorithm for multi-accelerator systems that aims to improve performance by analysis on directed acyclic graphs (DAG). It consists of two major phases: a task partitioning phase for determining the order of tasks based on mean computation and mean communication cost and a processor selection phase for assigning each task into an optimal processor with an insertion-based policy. HEFT shows satisfactory performance in both quality and cost of schedules in DAG-based task scheduling.

However, it does not consider the irregularity of tasks. If there is no data dependency between incoming tasks, which means the communication cost is zero, the HEFT algorithm will assign the highest priority to the task that has the minimum execution time. Like ranking-based scheduling methods, the basic idea of HEFT is to determine the order of tasks based on a specific heuristic and then schedule these tasks according to their order.

3.3.3 M. Handa et al. (2004)

M. Handa et al [49] proposed a dynamic scheduler with runtime resource allocation and task placement targeting multi-task co-execution on single FPGA-based reconfigurable platforms. They use a rectangle-based model to represent the fragmentation of the reconfigurable hardware resources on FPGAs as well as the tasks. Concerning determining the shape and size of each task model, three time-related factors are applied including the task arrival time, start time and execution time. Tasks are also labelled with priority, which is the preference of execution orders against others. Based on their modelling, they proposed a novel integrated method to schedule tasks. Instead of following the FIFO order to allocate resources then place tasks in priority queue for execution, they defer the scheduling decision of each task until it is necessary – right before it will be executed. In this approach, the priority of each tasks in the runqueue will influence both the resource

allocation and task placement simultaneously in an integrated way without the need to re-order the in-queue tasks further before execution.

Based on whether there are data dependences between tasks, they handle in-order processes and out-of-order processes by two separate scheduling policies when implementing the scheduler. When their scheduler addresses in-order processes, an additional runtime queue - *delete queue* is applied to record the task deletion events. This queue has the duty to release – the following tasks dependent on the current tasks will be released once the current task finish as shown in the delete queue.

Concerning the handling of out-of-order processes, there is no need to use such a queue to record whether a certain task is finished and no lock is needed. The ready task will be executed whenever there is sufficient hardware resources. This is determined by matching of rectangle-based task model to resource model. It is clear that handling such out-of-order tasks can result in much higher resource utilization than in order tasks.

They apply the *empty area of resource per time* metric, which represents the ratio of resource idle to measure the quality of scheduling decisions and then use the *ratio of delay time in total execution time* to measure the overall performance. On average, their approach achieves 10%-14% performance gain over the previous non-integrated FPGA multi-task scheduler.

3.3.4 F. Redaelli et al (2010)

F. Redaelli et al [80] propose a multi-FPGA scheduler by investigating the partially reconfigurable and the dynamically reconfigurable features of these systems.

In addition of previous work on multi-task scheduler targeting single FPGA systems, this work provides an efficient design framework to determine the best hardware configuration for workloads on multi-FPGA systems, including the number of FPGAs for each workload. The goal of the underlying method of this framework is to minimise the total execution time of each task by parallelising the tasks on multiple co-executed FPGAs, but still taking into account the trade off with the additional communication overhead. The best configuration in terms of number of FPGAs will be reached when multi-FPGA communication delay begin to dominate the execution time at some point.

Their scheduler follows a heuristic-based approach to make decisions and select tasks in an *just-in-time* manner. When there are multiple possible

resources to execute a ready task, the heuristic instead uses the *farthest placement* principle and uses an anti-fragmentation technique. It aims at providing a better solution space for future placements, as it is has been demonstrated that it is easier to place large tasks in the center of the FPGA [5]. The scheduler also considers the latency from data migration between multi-FPGA to determine whether to reuse resources. Other runtime techniques, including configuration prefetching and limited reconfiguration, are also applied to optimal the scheduling decision.

Chapter 4

COLAB: A Heterogeneity-Aware Scheduler for Asymmetric Chip Multi-core Processors

4.1 Introduction

Most processor chips are incorporated into embedded devices, such as smartphones and IoT sensors, which are by nature, energy limited. Therefore, energy efficiency is a crucial consideration for the design of new processor chips. Heterogeneous systems combine processors of different types to provide energy-efficient processing for different types of workloads. In central processors, single-ISA asymmetric multicore processors (AMPs) are becoming increasingly popular, allowing extra flexibility in terms of runtime assignment of threads to cores, based on which core is the most appropriate for the workload, as well as on the current utilisation of cores. As a result of this, efficient scheduling for AMP processors has attracted a lot of attention in the literature [74]. The three main factors that influence the decisions of a general purpose AMP scheduler are:

- ***Core sensitivity.*** Cores of different types are designed for different workloads. For example, in ARM big.LITTLE systems, big cores are designed to serve latency-critical workloads or workloads with Instruction Level Parallelism (ILP). Running other kinds of workloads on them

would not improve performance significantly while consuming more energy. *Therefore, it is critical to predict which threads would benefit the most from running on which kind of core.*

- ***Thread criticality.*** Executing a thread faster does not necessarily translate into improved performance. An application might contain *critical* threads, the progress of which determines the progress of the whole application and it is these threads that the scheduler needs to pay special attention to. *Therefore, it is essential to identify critical threads of an application and accelerate them as much as possible, regardless of core sensitivity.*

- ***Fairness.*** In multiprogrammed environments, scheduling decisions should not only improve the utilisation of the system as whole, but should also ensure that no application is penalised disproportionately. Achieving fairness in the AMP setting is non-trivial, as allocating equal time slices in round robin manner to each application does not imply the same amount work done for each application. *Therefore, it is critical to ensure that each application is able to make progress in a fair way.*

The research community has put considerable effort into tackling these problems. Prior research [48, 25, 60, 96, 36] has explored bottleneck and critical section acceleration, others have examined fairness [114, 104, 100, 68, 69], or core sensitivity [21, 66, 6]. More recent studies [64, 63, 85, 99, 61] have improved on previous work by optimizing for multiple factors. Such schedulers tuned for specific kinds of workloads – either single multi-threaded program or multiple single-threaded programs. Only one previous work, WASH [58], can handle general workloads composed of multiple programs, each one single- or multi-threaded, with potentially unbalanced threads, and with a total number of threads that may be higher than the number of cores. While a significant step forward, WASH only controls core affinity and does so through a fuzzy heuristic. Coarse-grain core affinity control means that it cannot handle core allocation and thread dispatching holistically to speed up the most critical threads. The latter means that WASH has only limited control over which threads run where, leaving much of the actual decision making to the underlying Linux CFS scheduler.

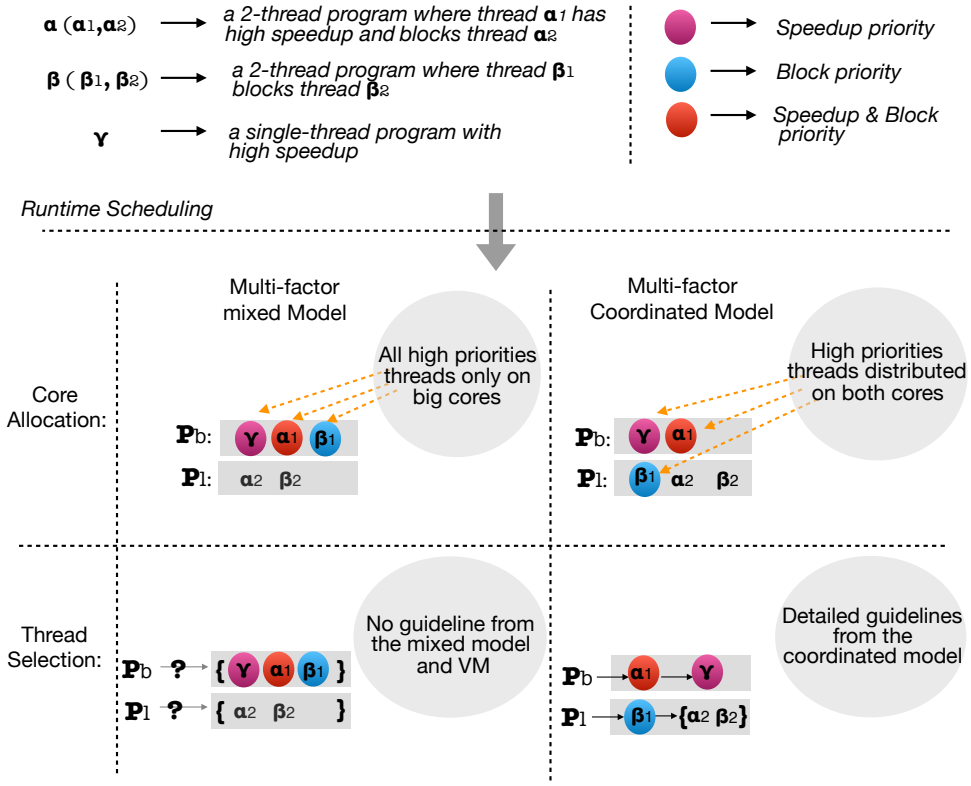


Figure 4.1: Motivating Example: Multi-threaded multiprogrammed workload on asymmetric multicore processors with one big core P_b and one little core P_l. The mixed model in the left hand side shows WASH decision and the collaborated model in the right hand side shows the proposed COLAB decision. Controlling only core affinity results in suboptimal scheduling decisions.

4.1.1 Motivating Example

Consider the motivating example shown in Figure 4.1 to demonstrate the problem. This AMP system has a high performance big core, P_b, and a low performance little core, P_l. Three applications, α, β, γ, are being executed. α and β have two threads. The first thread of each application, α_1 and β_1, blocks the second thread of their application, α_2 and β_2 respectively. γ is a single-threaded application. α_1 and γ enjoy a high speedup when executed on the big core, P_b. WASH [58], the existing state-of-the-art multi-factor heuristic, would be inclined to assign the high speedup thread and the two blocking threads to the big core. The thread selector of P_b has no information about the criticality of the threads assigned to it, so the order of execution depends on the underlying Linux scheduler.

A much better solution is possible if the scheduler could control both core allocation and thread selection in a coordinated, AMP-aware way. In this case, it maps the two threads that benefit the most from the big core, γ and α_1, to P_b, while it maps the other bottleneck thread, β_1, to P_l. This will not impact the overall performance of β. The thread selector knows β_1 is a bottleneck thread and executes it immediately. So, what the approach loses in execution speed for β_1, it gains in not having to wait for CPU time. Similarly, this coordinated policy guarantees that α_1 will be given priority over γ.

In this chapter, a novel OS scheduling policy, COLAB, is introduced for asymmetric multicore processors that can make such coordinated decisions. The proposed scheduler uses three collaborating heuristics to drive decisions about core allocation, thread selection, and thread preemption. Each heuristic optimizes primarily one of the factors affecting scheduling quality: core sensitivity, thread criticality, and fairness respectively. Working together, these multi-factor heuristics result in much better scheduling decisions.

COLAB is integrated inside the Linux scheduler module, replacing the default CFS policy for all application threads. It is evaluated on multiple big.LITTLE-like simulated systems running 36 distinct workloads, each workload being a random selection of PARSEC3.0 and SPLASH2 benchmarks. In almost all cases, COLAB was able to improve both turnaround time and throughput compared to the state-of-the-art and the Linux default. In the best case, where the workloads are mainly composed by synchronous-intensive workloads, turnaround time was 25% less under the proposed policy than under WASH and CFS.

The main contributions of this work are:

- The first AMP-aware OS scheduler targeting general multi-threaded multiprogrammed workloads.
- A set of collaborative heuristics for prioritizing thread based on core sensitivity, thread criticality, *and* fairness.
- Up to 25% and 21% lower turnaround time, 11% and 5% on average, compared to the Linux CFS and WASH scheduler.

The remainder of this chapter is presented as follows: Section 4.2 analyses the multiple runtime factors. The multi-factor collaboration is presented in Section 4.3. Section 4.4 shows the scheduling design and implementation. The experimental setup and results are presented in Section 4.5 and 4.6. Section 4.7 provides a summary of this chapter.

4.2 Runtime Factor Analysis

This section analyses the performance impact of multiple runtime performance factors and their relationships with different functional units in the scheduler. The heterogeneity-aware runtime scheduler is designed to address these performance problems in a coordinated way.

The high-level relationships between runtime performance factors and the functions which address them is shown in Figure 4.2. In order to achieve runtime collaboration, both core allocator and thread selector share information and account for all measured performance factors, including core sensitivity, bottleneck acceleration and fairness carefully as illustrated below:

4.2.1 Core Allocator

AMP-aware core allocators are mainly directed by the core sensitivity factor. Migrating a high speedup thread (with a large differential between big and little core execution time) from a little core to execute on a big core will generally provide more benefit than migrating a low speedup thread.

However, this heuristic is overly simplistic. Issues are revealed when the bottleneck factor is considered simultaneously on multiprogrammed workloads. Previous approaches [58] simply combine the calculation from bottleneck acceleration and predicted speedup together, but this can result in

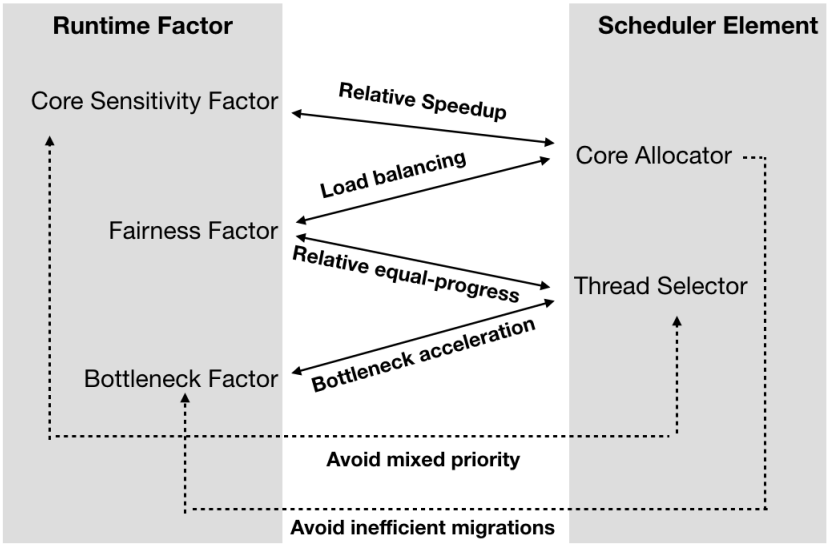


Figure 4.2: A diagram of how runtime performance factors are influenced by functions in the scheduling algorithm. Left hand side are the list of runtime factors and right hand side shows how the scheduling algorithm can do with them. The solid arrows represent how scheduling functions can benefit those runtime factor while the dotted arrows represent the possible negative influence from the scheduling functions to these runtime factors.

suboptimal scheduling decisions – both locking threads and high speedup threads may be accumulating in the runqueue of big cores as is illustrated in the motivating example in the Introduction. More intelligent core allocation decisions can be made by avoiding a simple combination of bottleneck acceleration and speedup – the overall schedule can benefit from a more collaborative execution environment where big cores focus on high speedup bottleneck threads, and little cores handle other low speedup bottlenecked threads without additional migration.

Furthermore, core allocators are designed to achieving relative fairness on AMPs by efficiently sharing heterogeneous hardware and avoiding idle resource as much as possible. Simply mapping ready threads uniformly between different type of cores can not achieve true load-balancing – the number of ready threads prioritized on different type of core is different and thus, a hierarchical allocation should be applied to guarantee the overall fairness, which avoids the need to frequently migrate threads to empty runqueues.

4.2.2 Thread Selector

The thread selector makes the final decisions on which thread will be executed during runtime. It is usually the responsibility of the thread selector to avoid bottlenecking by thread blocking. In a multi-thread multiprogrammed environment, multiple bottleneck threads from different programs may need to be accelerated simultaneously with constraint hardware resources. Instead of simply detecting the bottleneck threads and throwing all of them to big cores as previous bottleneck acceleration schedulers [58, 61, 60], the thread selector needs to make collaborative decisions – ideally, both big cores and little cores select bottlenecks to run simultaneously.

Core sensitivity is usually unrelated to the thread selector – whether a thread can enjoy a high speedup from a big core compared with a little core is unrelated to which runqueue it is on, or came from. Therefore the thread selector should separate thread priority caused by core sensitivity and solely base decisions on bottleneck acceleration. One exception is that when the runqueue of a big core is empty and the thread selector is invoked – the speedup factors from core sensitivity of ready threads should be considered only in this case. Big cores may even preempt the execution of little cores when necessary.

The final concern of thread selector is about fairness. Scaling the time slice of threads by updating the time interval of thread selector has been

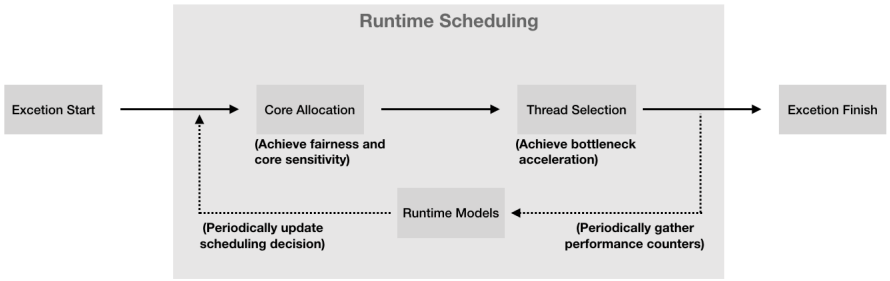


Figure 4.3: Flowchart of the scheduling process with a runtime feedback loop

shown to efficiently guarantee the equal progress [99] in multi-threaded single-program workloads and achieve fairness. In single-threaded multiprogrammed scenarios, complicated fairness formulation [64] has been proposed to guide the thread selector for precise decisions. Problems occur when targeting multi-threaded multi-programmed workloads. Simply keeping a thread-level equal progress is not enough to guarantee the multi-application level fairness – the thread selector should ensure the whole workload is in equal progress without penalizing any individual application. In fact, multi-bottleneck acceleration by both big and little cores does provide an opportunity for this - the thread selector makes the best attempt to keep fairness on all applications by accelerating bottlenecks from all of them and as soon as possible.

4.3 Multi-factor Runtime Collaboration

A coordinated multi-factor runtime collaboration is designed to address the problems detailed above, in which the core allocator and the thread selector collaborate to achieve high performance and high fairness, when compared to the state-of-the-art mixed multi-factor evaluator in WASH [58]. The flowchart of the proposed model is shown in Figure 4.3. Collaboration is facilitated by periodically labeling ready threads in two different categories, based on runtime models of speedup prediction and bottleneck identification:

4.3.1 Labels for Core Allocation

Threads with high predicted speedup between big and little cores will be labelled as high priority on big cores. Threads with both low predicted

speedup and blocking levels – non-critical threads – will obtain high priority on little cores (and low priority on big cores). Remaining threads obtain equal priority on either big or little cores – these threads can then be allocated freely to balance the load of cores.

4.3.2 Labels for Thread Selection

Threads with high blocking level will be labelled as high priority on local thread selection. The same priority will be given on these blocking threads whether the issuing cores are big or little, so the labels of thread selection do not distinguish the type of cores.The label nevertheless records the type of the current core – threads have priority to be selected by the same type of cores if there exists a core of the same type with an empty runqueue. Running threads on little cores are also labeled as they may be preempted to migrate and execute on big cores when suited, but running threads will never have priority over waiting ready threads.

4.3.3 Relative Equal Progress

Another important issue handled by the collaborative multi-factor model is to ensure relative equal-progress of threads. Instead of interfering with the priority and decisions of thread selection, the scheduler achieves equal progress of threads by applying the scaled time slice approach, based on the predicted speedup value of threads running on big cores. The slices of threads on big cores are relative shorter than on little cores. The thread selection function is triggered more often to swap executing threads on big cores, which guarantees the equal-progress of threads executed on all cores.

4.3.4 Multi-bottleneck Co-acceleration

After the above process, fairness, core sensitivity and bottleneck acceleration are represented by labels on threads can be handled by either the core allocator or the thread selector or both together in a fairness way. Based on this coordinated model, the core allocator and thread selector handle different priorities queues from the set of ready threads – their decisions are not greedy on a mixed multi-factor ranking like WASH, rather provide a collaborative schedule.

The runtime model periodically extracts the performance counters, which represents the current execution environment of multi-threaded multi- programmed workloads on the AMPs. The model then provides the updated runtime feedbacks, including the predicated speedup value and blocking counts. This information is attached to the threads and reported back to the multi-factor labeler for next round. The runtime model implementations is presented in the section below.

4.4 Scheduling Algorithm Design and Implementation

The COLAB scheduling algorithm described in this subsection is implemented on the GEM5 simulator [14], modifying the simulator and constructing interfaces between the Linux kernel v3.16 with the CFS scheduler.

4.4.1 Runtime Factors Implementations

The runtime multi-factor model is implemented by updating the main scheduler function `__sched__schedule()` of the Linux kernel by adding a thread process as described in section 3.2 above. A similar approach is followed by the WASH re-implementation when updating thread affinities.

Machine Learning based Speedup Prediction

Predicting the performance of threads on different core types is central for any scheduler targeting AMPs. This prediction uses an offline trained speedup model to estimate speedups online. This is a common approach in previous works [99, 58, 85].

Benchmarks from PARSEC3.0 are compiled all in single-program mode to construct the training set with two symmetric configurations, using either only little cores or only big cores. Firstly, all 225 performance counters of the simulated big cores and the relative speedup between the two configurations are recorded. Since on a real system, there will not be uniform high speed access to all performance counters simultaneously, Principal Component Analysis (PCA) technique [105] is applied to select the six performance counters with the largest effect on speedup modelling. They are then nor-

Table 4.1: Selected performance counters and Speedup Model

Selected GEM5 performance counters by PCAT		
Index	Name	Description [14]
A:	fp_regfile_writes	number of integer regfile writes
B:	fetch.Branches	number of branches that fetch encountered
C:	rename.SQFullEvents	number of times rename has blocked due to SQ full
D:	quiesceCycles	number of cycles quiesced or waiting for an interrupt
E:	dcache.tags.tagsinuse	cycle average of tags of dcache in use
F:	fetch.IcacheWaitRetryStallCycles	number of stall cycles due to full MSHR
G:	commit.committedInsts	number of instructions committed
Linear predictive speedup model		
2.6109+((0.0018*-0.185A)+(0.0259*0.187B)+(0.1047*0.194C)+(-0.023*0.238D)+(0.0492*-0.299E)+(-0.1388*-0.227F))/G		

malised to the number of committed instructions and linear regression is applied to build the final model, which is shown in Table 4.1.

Bottleneck Identification

On modern Linux systems synchronisation primitives are almost always implemented using kernel `futex`, regardless of the threading library used. Futex-based mechanisms use a single atomic instruction in user space to acquire or release the futex, if it is uncontested. Otherwise, it triggers a system call which forces the thread to sleep or wakes up sleeping threads, respectively.

This gives a convenient single point where the scheduler can monitor blocking patterns between threads. The first code addition is in `futex_wait_queue_me()` and `futex_lock_pi()`, right before the active thread starts waiting on a futex. It records the current time and store it in the `task_struct` of the thread. Followed by that, code is inserted in `wake_futex()` and `wake_futex_pi()`, right before the waiting task is woken up by the thread releasing the futex. There the length of the waiting period can be calculated and then accumulated in the `task_struct` of the thread releasing the futex. This way the scheduler will be able to measure the cumulative time each thread has caused other threads to wait. This is used as the metric of thread criticality for the rest of this chapter.

Speedup based Scale-slice Preemption

Although implementing the functions on Linux kernel by fully re-writing both the core allocator and thread selector, the underlining preemption mechanism of Linux is applying the virtual runtime *vruntime* in CFS with red-black tree

data structure. The vruntime of each task is its actual runtime normalized to the total number of running tasks. This vruntime-based preemption technique is designed to get multiple co-executed tasks all finished as soon as possible. Whenever a new task is enqueued, a preemption wake-up function is invoked to check whether the new coming task should preempt the current task by computing the difference in vruntime and comparing with a boundary. So the vruntime of each task can efficiently specify when its next timeslice would start execution on ideal multi-tasking CPU, where each task is ran at precise equal speed.

However, this is problematic on AMPs. Since different types of cores have different frequencies, speeds and instruction processing models (in-order or out-of-order), precise equal timeslice can not lead to equal progress of each task. Threads running on different types of cores should have different lengths of time slices in proportion to their performance variance to achieve equal-progress on AMPs.

The default preemption wake-up function `wakeup_ preempt_entity()` is updated in Linux kernel. The runtime speedup model is then applied to update the vruntime of the current task by dividing it by the thread's speedup value if the triggering core is a big core. The ensures relative equal progress.

4.4.2 Scheduling Algorithm Implementation

In brief, the default Linux CFS scheduler implemented in `sched/fair.c` contains an extensible hierarchy of scheduler modules, which encapsulate scheduling policy details. Scheduler modules contain hooks to functions that will be called when corresponding event occurs. Two main functions here are `pick_next_task_fair()` for choosing the most appropriate task eligible to run next and `select_task_rq_fair()` for choosing the most appropriate runqueue of a ready task.

The proposed COLAB scheduling algorithm is implemented by overriding the default `pick_next_task_fair()` and `select_task_rq_fair()` in the Linux kernel. The pseudo-code of COLAB scheduling algorithm is shown in Alg. 2. Similarly to commonly used Linux notations, *rq* is used to represent runqueue and *cur* is used to represent the current task of a core in the code. The main motivation of this algorithm design is to distribute the high speedup and bottleneck threads to be co-accelerated on multiple types of cores instead of accumulating on big cores as previous approaches and in such a way, to achieve multi-factor collaboration. The descriptions of the

Algorithm 1 Collaborative Multi-factor Scheduler targeting Asymmetric Multicore Processors

```
1: core_alloctor(thread_struct t){
2: if t.high_speedup then
3:    return rr_allocator(big_cores)
4: if t.low_speedup & t.low_block then
5:    return rr_allocator(little_cores)
6: return rr_allocator(cores)
7: }
8:
9: thread_selector(core_struct c){
10: if !empty(c.rq) then
11:    return max_block(c.rq)
12: if !empty(c.sched_domain.rq) then
13:    return max_block(c.sched_domain.rq)
14: if c.cpu_mask == big then
15:    return max_block(c.sched_domain_little.cur)
16: return idle
17: }
```

two main functions are listed below:

Hierarchical Core Allocator

When a thread is ready to be executed, whether it was just spawned or woken, the core allocator will be invoked to assign this thread to a core's runqueue. A hierarchical round-robin mechanism `rr_allocator()` is used to achieve relative load balancing and address the influence from the core sensitivity factor. Indicated by the speedup and blocking labels, threads are allocated to different core groups. Threads with high speedup will be assigned to big core clusters in round-robin (line 3). Low speedup and low blocking threads will only be assigned to little core clusters (line 5).

Remaining ready threads (usually with average speedup level and little blocking) will be relatively equally allocated to both core types by `rr_allocator(core)`. This final round-robin decision helps to keep both core clusters equally occupied and load balanced (line 8).

Biased-global Thread Selector

The thread selector is based on the principle of accelerating the most critical/blocking thread as soon as possible, shown in lines 10 to 21. The selector tries to choose a thread from the local runqueue first (lines 11-13). When there are no ready threads and migration is beneficial, the core triggers the migration of a candidate thread waiting in another runqueue. The thread with the highest blocking level will be selected. A default design principle of Linux CFS scheduler is followed here to reduce the overhead of accessing state in other runqueues - returning the best candidate thread from the local core group first (line 14-16). Further, big cores are allowed to select and preempt a running thread on a little core to accelerate it (line 17-19). Big cores are only allowed to go idle only when there is no ready thread left (line 20) - for instance, little cores are not allowed to preempt a big core's execution.

In summary, the thread selector can still access all other runqueues when necessary, but it is biased to access neighbouring ones first. Note that the relative equal-progress for achieving fairness is addressed by the scale-slice preemption checker instead of the thread selector – each thread is given a maximum time slice relative to its expected performance on the asymmetric core. Further, cache efforts or interference should also be addressed by thread selector in the future work when targeting complicated architectures with

shared LLC or memory.

4.5 Experimental Setup

4.5.1 Experimental Environment

The experiments are run on GEM5, which simulates an ARM big.LITTLE-like architecture. The big cores are similar to out-of-order 2 GHz Cortex A57 cores, with a 48 KB L1 instruction cache, a 32 KB L1 data cache, and a 2 MB L2 cache. The little cores are similar to in-order 1.2 GHz Cortex A53 cores, with a 32 KB L1 instruction cache, a 32 KB L1 data cache, and a 512 KB L2 cache. Four distinct hardware configurations are evaluated. Two had balanced numbers of big and little cores, one with two big and two little cores (2B2S) and one with four big and four little ones (4B4S). The other two had different numbers of big and little cores, one with two big and four little cores (2B4S) and one with four big cores and two little cores (4B2S). The OS is Linux v3.16. The kernel is cross-compiled with gcc v5.4.0, while benchmarks inside the emulated environment are compiled with gcc v4.8.2.

The simulated environment is chosen to make it easier to evaluate the proposed approach on multiple different hardware configurations. While this experiment targeted simulated ARM cores, the underlying general procedure and model can be implemented on any real processor as long as they provide enough hardware performance monitor units (PMU). All hardware counters used by the model are supported by the real ARM Cortex-A57/A53 [3] PMU.

4.5.2 Workloads Composition

The system are evaluated using 15 different benchmarks (Table 4.2), pulled from PARSEC3.0 [11] and from SPLASH2 [106] applied to generate the tested workloads. [1] To keep the simulation time reasonably short, all benchmarks use the *simsmall* inputs. The benchmarks are grouped based on two criteria: a) synchronization intensity and b) communication vs computation intensity.

[1] The applied version of GEM5 cannot simulate all PARSEC3.0 benchmarks on ARM. On top of *canneal* and *raytrace* that other researchers have also failed to build [38, 101]. Among the SPLASH2 benchmarks, *cholesky* and *volrend* depended on huge input data files that did not fit on the hard drive image. Other benchmarks not used in the experiments includes *barnes*, *radiosity*, *facesim*, *x264*, *vips* and *streamcluster* because their runtime is prohibitively long even for *simsmall* inputs.

Table 4.2: Benchmarks categorization [12, 106, 92]

Name	Sync. Intensity	Comm/Comp Intensity
blackscholes	low	high
bodytrack	medium	high
dedup	medium	high
ferret	high	medium
fluidanimate	very high	low
freqmine	high	high
swaptions	low	low
radix	low	high
lu_ncb	low	low
lu_cb	low	low
ocean_cp	low	low
water_nsquared	medium	medium
water_spatial	low	low
fmm	medium	low
fft	low	high

Table 4.3: Multi-programmed Workloads Compositions

Synchronization-intensive VS Non-synchronization-intensive Workloads			
Index	Workload Composition	Synchronizations	Threads
Sync - 1	water_nsquared - fmm	intensive	4
Sync - 2	dedup - fluidanimate	intensive	18
Sync - 3	water_nsquared - fmm - fluidanimate - bodytrack	intensive	9
Sync - 4	dedup - ferret - fmm - water_nsquared	intensive	20
NSync - 1	water_spatial - lu_cb	non-intensive	4
NSync - 2	blackscholes - swaptions	non-intensive	16
NSync - 3	radix - fft - water_spatial - lu_cb	non-intensive	8
NSync - 4	blackscholes - ocean_cp - lu_ncb - swaptions	non-intensive	20
Communication-intensive VS Computation-intensive Workloads			
Index	Workload Composition	Comm/Comp	Threads
Comm - 1	water_nsquared - blackscholes	Communication-intensive	4
Comm - 2	ferret - dedup	Communication-intensive	16
Comm - 3	water_nsquared - fft - radix - bodytrack	Communication-intensive	9
Comm - 4	blackscholes - dedup - ferret - water_nsquared	Communication-intensive	20
Comp - 1	water_spatial - fmm	Computation-intensive	4
Comp - 2	fluidanimate - swaptions	Computation-intensive	17
Comp - 3	lu_ncb - fmm - water_spatial - lu_cb	Computation-intensive	8
Comp - 4	fluidanimate - ocean_cp - lu_ncb - swaptions	Computation-intensive	20

Random-mixed Multi-programmed Workloads					
Index	Workload Composition	Threads	Index	Workload Composition	Threads
Rand - 1	lu_cb - dedup	19	Rand - 6	water_spatial - fmm - fft - fluidanimate	21
Rand - 2	lu_ncb - bodytrack	10	Rand - 7	fmm - water_spatial - ferret - swaptions	20
Rand - 3	ferret - water_spatial	9	Rand - 8	water_spatial - water_nsquared - ferret - freqmine	17
Rand - 4	ocean_cp - fft	8	Rand - 9	blackscholes - bodytrack - dedup - fluidanimate	55
Rand - 5	freqmine - water_nsquared	6	Rand - 10	lu_cb - lu_ncb - bodytrack - dedup	53

The insight information of each benchmark is extracted from its source papers [12, 106, 92]. For each group, workloads are randomly generated with variable numbers of benchmarks and threads. These workloads provide the opportunities to investigate the behavior of the three scheduling policies under different scenarios. There are also 10 workloads generated with random benchmarks from all groups to explore the general case of scheduling for an AMP system. Table 4.3 shows the selected workloads. For all of them, the experiment starts from a checkpoint taken after all benchmarks have completed their initialization.

Each individual result represents the average over two simulations with different core orders - either big cores first or little cores first. Because there are only two types of cores and practical AMPs either configure big or little core first without trying to arrange them by mixing up. Small variations in the initial state of the system might have a significant effect on scheduling decisions and thus performance. For the Linux scheduler in particular, the order of starting benchmarks will decide which benchmarks will be initially assigned to big and little cores. By varying the initial state and measuring average runtimes over multiple simulations, the experiment minimizes the effect of randomness on the evaluation. The WASH and COLAB provide more finer grained control on thread scheduling on AMPs than Linux, which lead to deterministic and repeatable solutions on different core orders.

4.5.3 Schedulers

COLAB is evaluated by comparing it against the Linux CFS scheduler [75] and a state-of-the-art realistic scheduler based on WASH [58]. CFS is the default Linux scheduler and it provides fairness while trying to maximize the overall CPU resource utilization. WASH is re-implemented driven with a core sensitivity model that fits the simulated system and it is used for controlling all application threads.

4.6 Results

4.6.1 Single-programmed Workloads

Much of the research on AMP scheduling focuses on single-programmed workloads. In this context, fairness and load balancing are not important, the

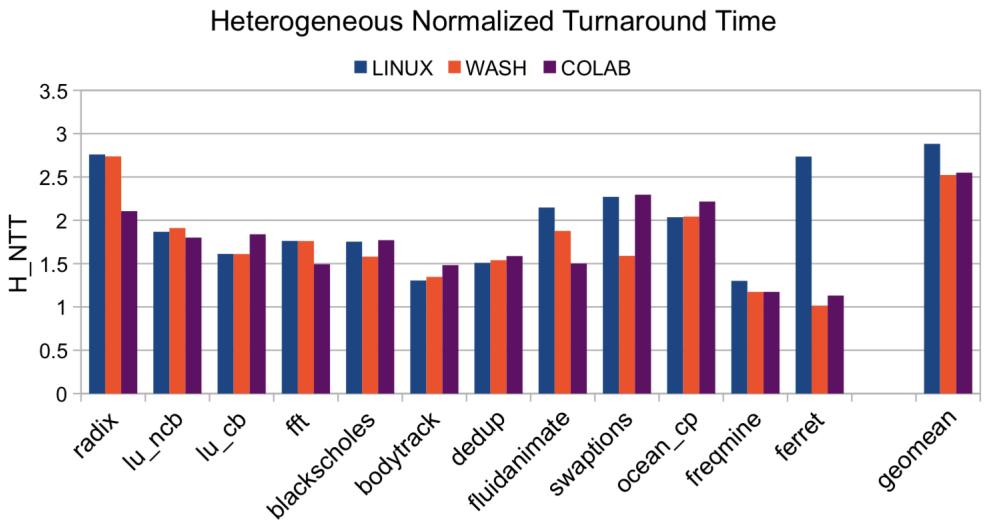


Figure 4.4: Heterogeneous Normalized Turnaround Time (H_NTT) of single program workloads on a 2-big 2-little system. Lower is better

focus is on core sensitivity and bottleneck acceleration. In this section, how COLAB fares under this scenario is examined. Figure 4.4 shows Heterogeneous Normalized Turnaround Time for the multi-threaded benchmarks when executed alone on a 2-big-2-little hardware configuration. For each configuration and benchmark, there are three bars presented, Linux(blue), WASH(red) and COLAB(violet). Three SPLASH2 benchmarks *fmm*, *water_nsquared* and *water_spatial* do not support more than 2 threads with *simsmall* input size on GEM5, so those 2-threaded benchmarks are not presented: scheduling them optimally for performance is trivial.

The AMP-agnostic Linux scheduler is inappropriate for most benchmarks. COLAB improves H_NTT by up to 58% and by 12% on average. The best result from COLAB relative to Linux is for *ferret*. Most computation happens in a pipeline pattern but its stages are not balanced. AMP-aware schedulers take advantage of that by scheduling the longest stages, the bottleneck threads, on big cores. As a result, COLAB does only 13% worse than running on a system *with four big cores*, while CFS executes the benchmark 173% slower.

Compared to WASH, COLAB achieves its best result for *fluidanimate*. Previous work [12] has shown that *fluidanimate* has around 100x more lock-

based synchronizations than other PARSEC applications. The collaborative core allocation and thread selection policy is much better than WASH at prioritizing bottleneck threads. As a result, COLAB reduce turnaround time by 30% compared to Linux and 20% compared to WASH.

In some cases, such as *bodytrack*, *lu_ncb*, or *freqmine*, AMP-awareness has little effect on performance. Such benchmarks split work dynamically between threads. As a result, all threads have the same core sensitivity and the application adapts automatically to asymmetries in processing speed. Any AMP-aware scheduling policy, whether WASH or COLAB, will offer no benefit while introducing overhead. Such behavior was also apparent in WASH [58]. The pipeline benchmark *dedup* has five stages to stream the input set. When the number of threads is greater than the number of cores that can be run, both heterogeneous-aware schedulers can not service the excess threads in time, resulting in a certain impact on overall system performance.

There is only one case where COLAB performs significantly worse than WASH. For *swaptions*, COLAB performs as well as the AMP-agnostic Linux scheduler while WASH improves turnaround time by 31%. This is because the bottleneck threads of *swaptions* are core insensitive while the non-bottleneck threads are core sensitive. This being the ideal case for WASH, it improves turnaround time while COLAB fails to do the same.

On average, WASH and COLAB perform similarly well and improve performance by 12% compared to Linux when handling single program workloads. This is a limited scenario, with no need for fairness and a simple decision space. COLAB was not expected to perform much better than the state-of-the-art, doing as well as it is a positive result.

4.6.2 Multi-programmed Workloads

The main aim of the COLAB scheduler is to target workloads of multiple multi-threaded programs, which represents the most general case for CPU scheduling. In this section, the performance of COLAB is evaluated in this setting. Overall, it is able to outperform both the Linux CFS and WASH when there is room for improvement. This is particularly true when there are a limited number of big cores and/or many communication-intensive benchmarks. In such cases, the scheduler needs to consider *at the same time* both core affinity and thread bottlenecks. COLAB can do that, while CFS and WASH cannot, leading to significant performance improvements. In the rest

of this subsection, the behaviour of COLAB is examined under four different hardware configurations (2B2S, 2B4S, 4B2S, 4B4S) for the five different classes of workloads shown in Table 4.3.

Synchronisation-intensive vs Synchronisation Non-intensive workloads

The synchronisation-intensive group contains workloads where all programs have high synchronisation rates. Because of this, these workloads are expected to have a large number of bottleneck threads, so COLAB should be able to schedule them better than CFS and WASH. Conversely, synchronisation non-intensive workloads should provide few opportunities for COLAB to improve on CFS and WASH.

Figure 4.5 shows how well the three schedulers perform on average for each workload class and hardware configuration. The top plot shows the average H_ANTT while the bottom plot shows the average H_STP. The left half of each plot contains the results for the synchronization-intensive (*Sync*) workload class, while the right half is the synchronization non-intensive (*N_Sync*) workload class.

The results agree with the expectations. COLAB improves the turnaround time of *Sync* workloads by around 15% and 4% on average compared to CFS and WASH, respectively. Hardware configurations with low core counts, such as 2B2S, favors COLAB. It can reduce turnaround time by up to 20% over CFS and by up to 16% over WASH. With fewer cores, the pressure from co-executed applications rises and properly balancing bottleneck acceleration and core sensitivity across multiple programs becomes increasingly difficult. WASH places all bottleneck threads onto the big cores, which results in these threads having to wait for CPU time in busy run queues, ending up with only 3% of performance improvement over Linux. COLAB handles these bottleneck threads in a more holistic way, improving turnaround time by 20% and system throughput by 27%, compared to Linux.

As for *N_Sync* workloads, there are few bottleneck threads to be accelerated, making scheduling decisions much easier. As a result, both COLAB and WASH perform similarly to Linux, with COLAB improving average turnaround time by 6% and average system throughput by 12% compared to Linux.

An interesting point is that COLAB does significantly better (10% and 15% improvement on turnaround time) than WASH and Linux for *N_Sync*

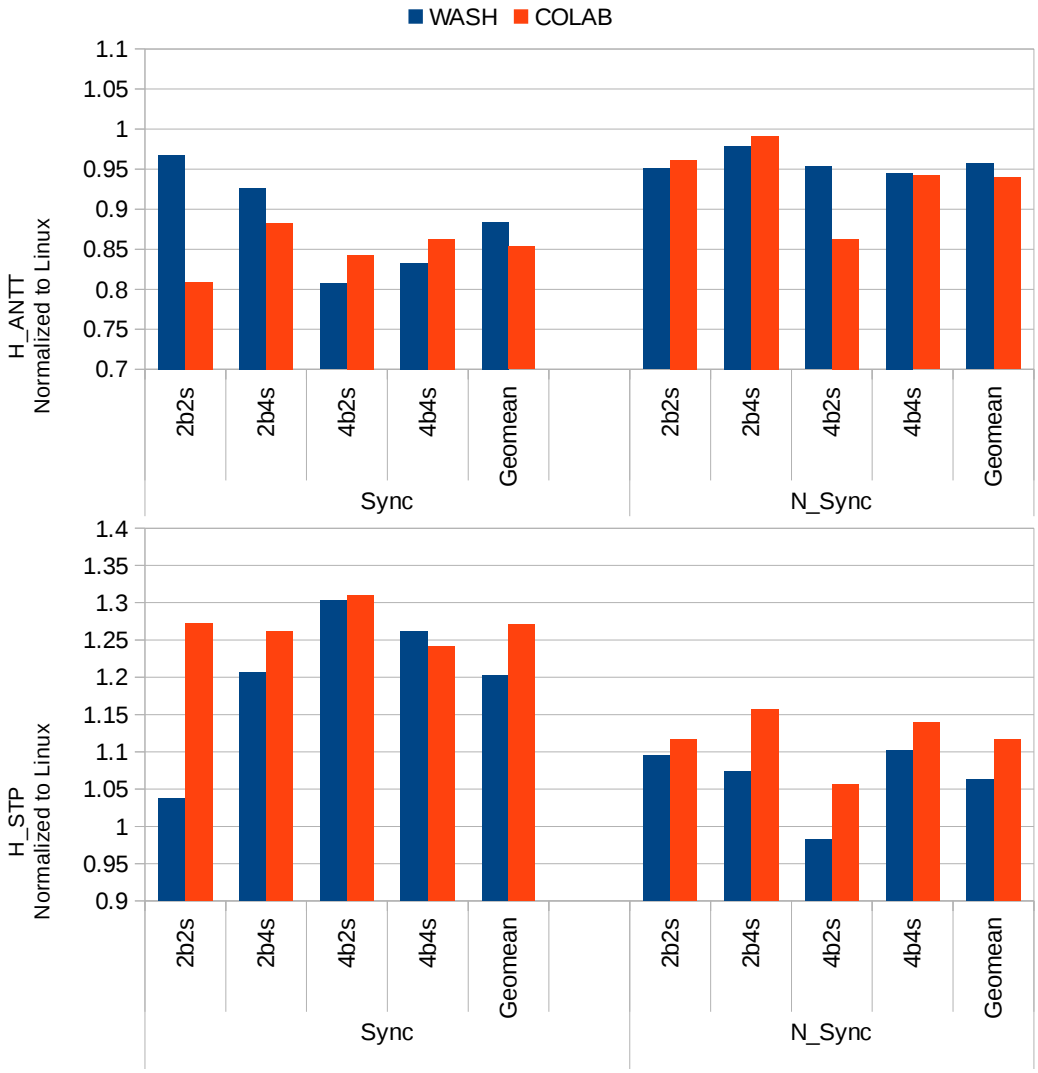


Figure 4.5: Heterogeneous Average Normalized Turnaround Time (H_ANTT) and Heterogeneous System Throughput (H_STP) of Synchronization-Intensive and Non-Synchronization-Intensive Workloads. All results are normalized to the Linux CFS ones. Lower is better for H_ANTT and higher is better for H_STP.

workloads on the 4B2S configuration. In this case, where there are sufficient big core resources without enough critical threads, WASH keeps migrating predicted critical threads on big cores even when there is no actual need. However, COLAB will make intelligent decisions by keeping relatively more threads on little cores, which gives more chance for big cores to execute the limited *really critical* threads as soon as possible.

Communication-intensive vs Computation-intensive workloads

When handling programs with high communication-to-computation ratios, bottleneck threads are likely to arise and accelerating them is critical. This is an ideal scenario for COLAB. On the other hand, workloads with little communication are easier to schedule, so CFS and WASH should do reasonably well, leaving little space for improvement.

Figure 4.6 shows the evaluation results for these two classes of workloads, *Comm* and *Comp*. Both COLAB and WASH improve over the Linux scheduler for communication-intensive workloads. They, however, offer different advantages on different hardware configurations. COLAB distributes the bottleneck threads to both big and little cores which is extremely important when having only two big cores (2B2S, 2B4S). COLAB improves the turnaround time by up to 21% compared to Linux and 15% compared to WASH on the 2B4S configuration. When more big cores are available, WASH does better as it keeps all bottleneck threads on big cores. On these configurations, WASH improves turnaround time by up to 18% over Linux (on the 4B4S configuration) and up to 10% over COLAB (on the 4B2S configuration). On average, COLAB reduces turnaround time by around 12% compared to Linux and 1% compared to WASH for the communication-intensive workload class.

Figure 4.6 also confirms that there are few opportunities for better scheduling with computation-intensive workloads. Still, COLAB does better than WASH and Linux. Its turnaround time and system throughput are improved by around 10% and 15%, respectively, compared to Linux and 5% compared to WASH. This is, again, due to a fact that multiple bottlenecks are distributed both to big and little cores, which results in more efficient use of the available hardware resources for the few bottlenecks that are present.

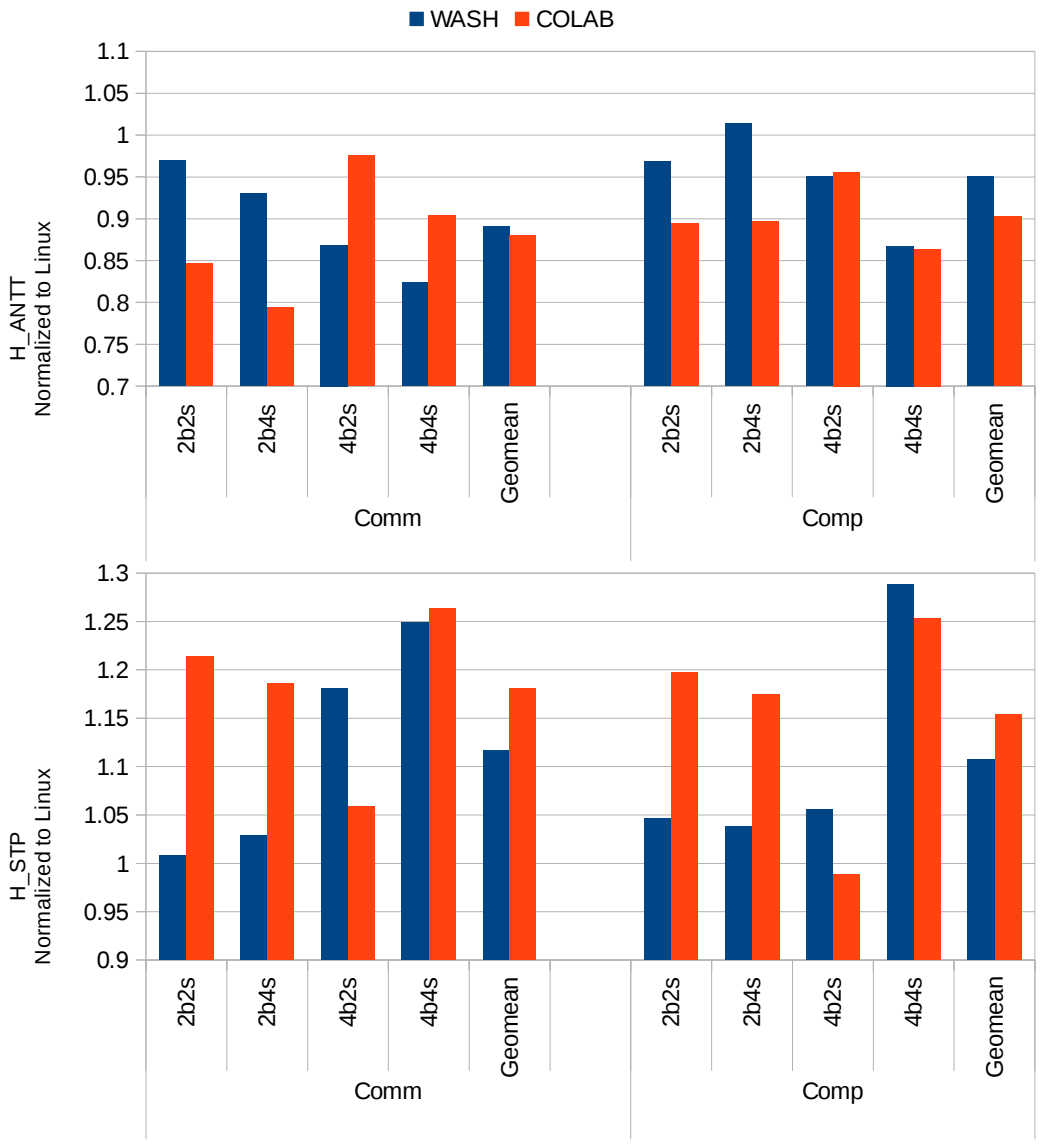


Figure 4.6: Heterogeneous Average Normalized Turnaround Time (H_ANTT) and Heterogeneous System Throughput (H_STP) of Communication-Intensive and Computation-Intensive Workloads. All results are normalized to the Linux CFS ones. Lower is better for H_ANTT and higher is better for H_STP.

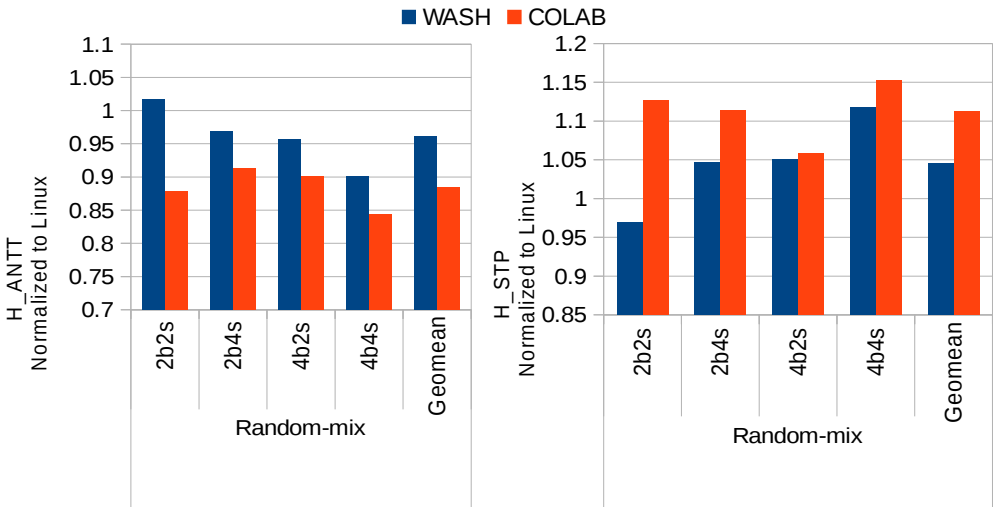


Figure 4.7: Heterogeneous Average Normalized Turnaround Time (H_ANTT) and Heterogeneous System Throughput (H_STP) of 2-programmed and 4-programmed Workloads. All results are normalized to the Linux CFS ones. Lower is better for H_ANTT and higher is better for H_STP.

Mixed workloads

Mixed workloads represent the general case of different applications with different needs, affinities, and communication patterns competing for the same cores. Figure 4.7 shows the average evaluation results for 10 such workloads. COLAB performs very well for these workloads: more diverse programs mean more asymmetry, more bottlenecks, more critical threads, and more potential for acceleration. The proposed collaborative multi-factor scheduling method carefully balancing all scheduling aims (core sensitivity, thread criticality and fairness) leads to a significant performance gain against WASH and Linux. COLAB improves turnaround time and system throughput by 12% and 11% compared to Linux and 8% and 7% compared to WASH in average.

Thread and program count

The experimental results are further grouped based on thread and program count to examine their impact on the behavior of each scheduler. Figure 4.8 shows the performance of all schedulers both for workloads with a low thread count (less than the core count for that hardware configuration) and for workloads with a high thread count (at least double higher than the maximum core count). Both COLAB and WASH perform significantly better than Linux for workloads with a low number of threads. Fewer threads make it easier to identify bottleneck threads and give them the resources they need - either by migrating them to big cores (WASH and COLAB) or by prioritizing them on little cores (COLAB). With limited big core resources, COLAB does much better than WASH since it distributes bottleneck threads on all available cores, avoiding overloading the few big cores and keeping the little cores idle. COLAB outperforms Linux by up to 25% (2B4S) and WASH by up to 21% (2B4S) on turnaround time. On average, COLAB improves turnaround time and system throughput by around 20% and 35% compared to Linux and around 8% and 11% compared to WASH for workloads with a low number of threads.

For workloads with a high thread count, neither Linux nor WASH are able to improve much on Linux. Overloading the system with threads means that, regardless of where threads are placed, cores will have long runqueues. COLAB and WASH increase the management overhead, including more frequent thread migrations. With little benefit from such better management, the management overhead leads to performance degradation. Of the two

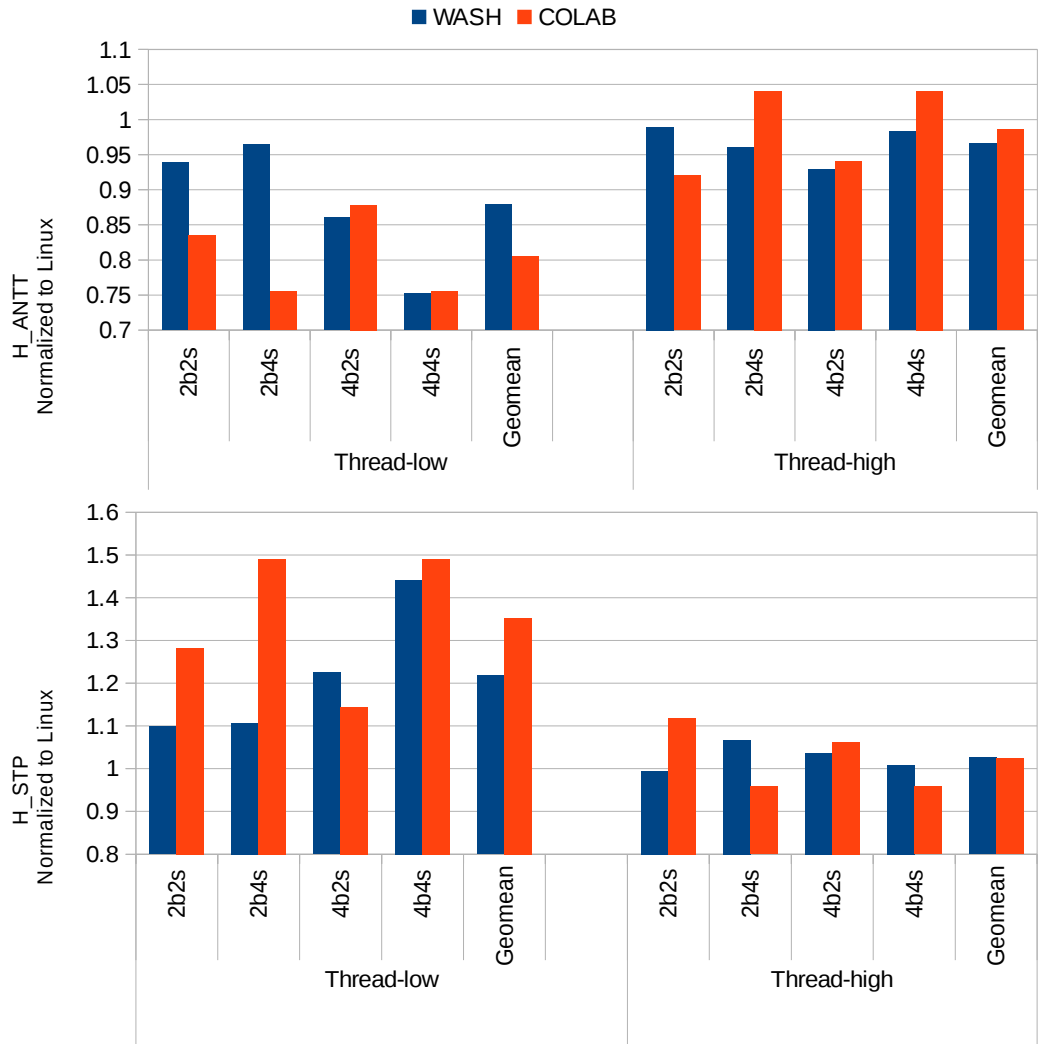


Figure 4.8: Heterogeneous Average Normalized Turnaround Time (H_ANTT) and Heterogeneous System Throughput (H_STP) of low number of application threads and high number of application threads Workloads. All results are normalized to the Linux CFS ones. Lower is better for H_ANTT and higher is better for H_STP.

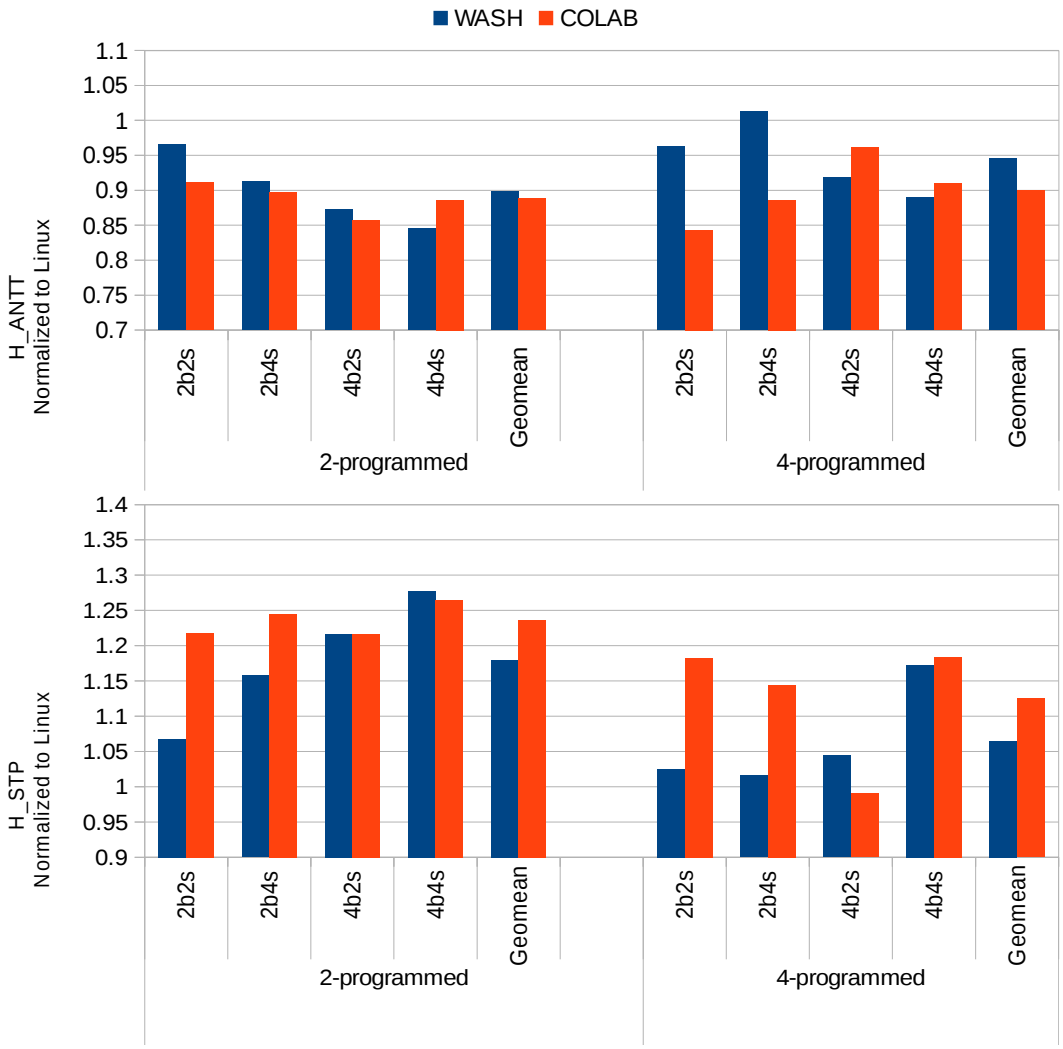


Figure 4.9: Heterogeneous Average Normalized Turnaround Time (H_ANTT) and Heterogeneous System Throughput (H_STP) of 2-programmed and 4-programmed Workloads. All results are normalized to the Linux CFS ones. Lower is better for H_ANTT and higher is better for H_STP.

heterogeneity-aware schedulers, COLAB, with its scale-slice technique, more frequently migrates threads, which results in a slightly worse performance than WASH. On average, COLAB improves turnaround time and system throughput by less than 2% and 3% compared to Linux, while WASH slightly outperforms COLAB by 2% on turnaround time and 0.2% on system throughput.

There is a similar picture considering workloads with different number of programs in them. Figure 4.9 shows the performance of all schedulers for 2-programmed and 4-programmed workloads. As in the case of high and low thread counts, increasing the number of co-executed programs gives higher pressure on the scheduler, increasing the waiting time of threads in runqueues and reducing the direct benefit of migration between waiting threads. But more programs also cause more bottlenecks and provide new opportunities for co-acceleration instead of only increasing data-parallel threads.

By intelligently distributing bottleneck threads from different programs between big and little cores, COLAB faces less problems than WASH from the pressure of increasing programs.

As a result, both COLAB and WASH outperform Linux by more than 10% on 2-programmed workloads on turnaround time and COLAB can keep the 10% performance gain also on 4-programmed workloads, while WASH reduced to only have 5% performance gain on 4-programmed workloads. As for system throughput, COLAB improves by 23% and 12% on 2-programmed and 4-programmed workloads compared to Linux while improves by 5% and 6% on 2-programmed and 4-programmed workloads compared to WASH.

4.6.3 Summary of Experiments

The experiments show that the state-of-the-art heterogeneous-aware WASH scheduler struggles to make better scheduling decisions that the Linux scheduler for a number of different types of workloads and configurations: synchronization-intensive workloads, computation-intensive workloads, low threads number workloads, high program number workloads, mixed multi-class workloads and limited big cores configurations. Trying to handle both core sensitivity and bottleneck acceleration through thread affinity alone may lead to too many threads assigned to big cores. Instead, COLAB handles the two optimization aims separately. It assigns on big cores only threads which run significantly faster on them and prioritizes running bottleneck threads regardless of their thread affinity. This leads to improved turnaround time,

higher throughput, and better use of the processor resources compared to both Linux and WASH. In summary from all experiments, COLAB improves turnaround time and system throughput by 11% and 15% compared to Linux and by 5% and 6% compared to WASH.

4.7 Conclusion

In this chapter, a novel **COLAB** scheduling framework is presented that targets multi-threaded multiprogrammed workloads on asymmetric multicore processors (AMPs) which occupy a significant part of the processor market today, especially in embedded systems. COLAB is the first general-purpose scheduler that simultaneously tries to optimize all three factors that affect the AMP scheduling - core affinity, thread criticality, and scheduling fairness. By making *collaborative* decisions on *core affinity* (in terms of selecting the most suitable core for a given thread of a workload) and *thread selection* (in terms of selecting which thread from a runqueue of a given core to run next), COLAB is able to improve on the state-of-the art WASH, as well as on the Linux CFS, which consider these two decisions in isolation.

Experimental results have demonstrated on a number of different workloads comprised of benchmarks taken from the state-of-the-art parallel benchmark suites PARSEC3.0 and SPLASH-2, simulating a number of different AMP configurations using the well-known GEM5 simulator, that the COLAB scheduler outperforms both WASH and Linux CFS scheduler by up to 21% and 25%, respectively, in terms of turnaround time (5% and 11% on the average). It also demonstrates improvements of 6% and 15% in terms of system throughput on the average. This demonstrates the applicability of the proposed approach in realistic scenarios, allowing better execution times for parallel workloads on AMP processors without additional effort from the programmer.

Chapter 5

SupCOLAB: A Heterogeneity-aware Data Partitioner and Scheduler for Supercomputers

5.1 Introduction

Large-scale workloads and algorithms usually have custom designs and implementations targeting specific supercomputers. This brings new opportunities to accelerate the system performance by not only efficiently scheduling the parallel threads during runtime as what COLAB has achieved, but also by customised partitioning of the original dataflow to exploit the unique features of given heterogeneous hardware resources. Under carefully data partitioning, the scheduler can give much more accurate decisions on assigning running threads to suitable resources. Having dealt with the complexity in the data partitioning, a simple scheduling model, such as the greedy approach, can result in an efficient solution. This also has the benefit of scaling well when targeting modern supercomputers with tens of millions of cores – the greedy scheduler can scale and work on massive solution space with the lowest system overhead.

A concrete study on addressing a representative type of large-scale workload, the k-means algorithm based workload, has been performed on the world-leading Sunway Taihulight supercomputer [43] to demonstrate the advantage of applying the heterogeneity-aware data partitioning and scheduling approach on real applications, named as SupCOLAB. It has shown significant

benefits of not only improving the system performance, but also achieving more scalability than the state-of-the-art.

The rest of this chapter is structured as follows: Section 5.2 shows the problem definition of k-means and discusses some state-of-the-art approaches to addressing k-means based large-scale workload on modern supercomputers. Section 5.3 presents the SupCOLAB data partitioner as a concrete study of k-means on Sunway Taihulight supercomputer. Section 5.4 shows the SupCOLAB scheduling algorithm. The experimental setup and results are shown in the sections 5.5 and 5.6.

5.2 A Case Study: K-Means

K-means is a well-known clustering algorithm, used widely in many AI and data mining applications, such as bio-informatics [8, 57], image segmentation[27, 55], information retrieval [93] and remote sensing image analysis[65].

The purpose of the k-means clustering algorithm is to find a group of clusters to minimize the mean distances between samples and their nearest centroids. Formalized, given n samples,

$$\mathcal{X}^d = \{x_i^d \mid x_i^d \in R^d, i \in \{1, ..., n\}\}$$

where each sample is a d-dimensional vector $x_i^d = (x_{i1}, \ldots, x_{id})$ and u is used to index the dimensions: $u \in \{1 \ldots d\}$. The object is to find k d-dimensional centroids $\mathcal{C}^d = \{c_j^d \mid c_j^d \in R^d, j \in \{1 \ldots k\}\}$ to minimize the object $\mathcal{O}(\mathcal{C})$:

$$\mathcal{O}(\mathcal{C}) = \frac{1}{n} \sum_{i=1}^{n} \mathrm{dis}(x_i^d, c_{a(i)}^d)$$

Where $a(i) = arg\ min_{j \in \{1 \ldots k\}} \mathrm{dis}(x_i^d, c_j^d)$ is the index of the nearest centroid for sample x_i^d, $dis(x_i^d, c_j^d)$ is the *Euclidean* distance between sample x_i^d and centroid c_j^d:

$$\mathrm{dis}(x_i^d, c_j^d) = \sqrt{\sum_{u=1}^{d} (x_{iu} - c_{ju})^2}$$

In the literature, several methods have been proposed to find efficient solutions [76, 77, 35, 31, 90, 19]. While the most popular baseline is still the

Lloyd algorithm [72], which is composed by repeating the basic two steps below:

$$1.: \ a(i) = arg\ min_{j \in \{1...k\}}\ dis(x_i^d, c_j^d)\ (Assign)$$

$$2.: \ c_j^d = \frac{\sum_{arg\ a(i)=j} x_i^d}{|arg\ a(i) = j|} \quad (Update)$$

An initial set of centroids also need to be chosen. The initial centroids usually randomly selected from the original datasets to avoid empty clusters.Note that those notations here are mainly from previous works by Hamerly [47], Newling and Fleuret [76]. Customised notations are only applied when needed. The first step above is to assign each sample into the nearest centroid according to the *Euclidean* distance. The second step is to update the centroids by moving them to the mean of their assigned samples in the d-dimensional vector space. Those two steps are repeated until each c_j^d is fixed.

5.2.1 Parallel K-Means

k-means algorithm has been widely implemented in parallel architectures with shared and distributed memory using either SIMD or MIMD model targeting on multi-core processors [34, 46, 16], GPU-based heterogeneous systems [115, 71, 98], clusters of computer/cloud [30, 51].

Commonly, l is used to index the processors (computing units) $\mathcal{P}$ ($\mathcal{P} = \{P_l\}, l \in \{1 \dots m\}$), and m is used to denote the total number of processors applied. The dataset $\mathcal{X}^d$ is partitioned uniformly into m processors. Compared with the basic *Lloyd* algorithm, each processor only assigns a subset ($\frac{n}{m}$) of samples from the original set $\mathcal{X}^d$ before the *Assign* step. Then the *Assign* step is finished in parallel by m processors. To formalize the steps, there is:

$$1.1 : P_l \leftarrow x_i^d, i \in (1 + (l-1)\frac{n}{m}, l\frac{n}{m})$$

$$1.2 : \forall l \in (1, m), P_l : \ a(i) = arg\ min_{j \in \{1...k\}}\ dis(x_i^d, c_j^d)$$

Message Passing Interface (MPI) library is mostly applied in common multi-core processor environments to facilitate communication between computing units. Performance nearly linearly increases with the limited number of processors as the communication cost between processes can be ignored in the non-scalable cases, as demonstrated in [34]. Similarly, the *Update* steps

are finished by m processors in parallel through MPI as well. Processors should communicate with each other before the final c_j^d can be updated.

5.3 Large-scale Data Partitioning on K-Means

This section presents how the proposed SupCOLAB approach efficiently partitions the large-scale dataflow of k-means algorithm targeting the given supercomputer, Sunway Taihulight, in a heterogeneity-aware way. This includes a multi-level dataflow partitioning method targeting the hierarchical hardware resources and storage limitations and then a self-aware automatic data partitioner. An automatic hyper-parameter detector is also been proposed to address on dataflows without pre-knowledge, for which the goal number of clusters (the k value) is not given by default. The partitioned workflow can be easily processed on the given supercomputer, exploit the fully advantages from the heterogeneous hardware resources and show breakthrough large scalability on tens of millions of co-executed cores.

5.3.1 Hierarchical Multi-level Partitioning

SupCOLAB explores the hierarchical parallelism on the heterogeneous many-core architecture to achieve efficient large-scale *k-means* on the Sunway supercomputer. It demonstrates the proposed scalable methods on three parallelism levels by how the data is partitioned.

- Level 1 - $DataFlow$ Partitioning: Store a whole sample and k centroids on single-CPE
- Level 2 - $DataFlow$ and $Centroids$ Partitioning: Store a whole sample on single-CPE whilst k centroids on multi-CPE
- Level 3 - $DataFlow$, $Centroids$ and $Dimensions$ Partitioning: Store a whole sample on multi-CPE whilst k centroids on Multi-CG and d dimensions on Multi-CPE

An abstract graph of how the data is partitioned into multiple levels is presented in Figure 5.1.

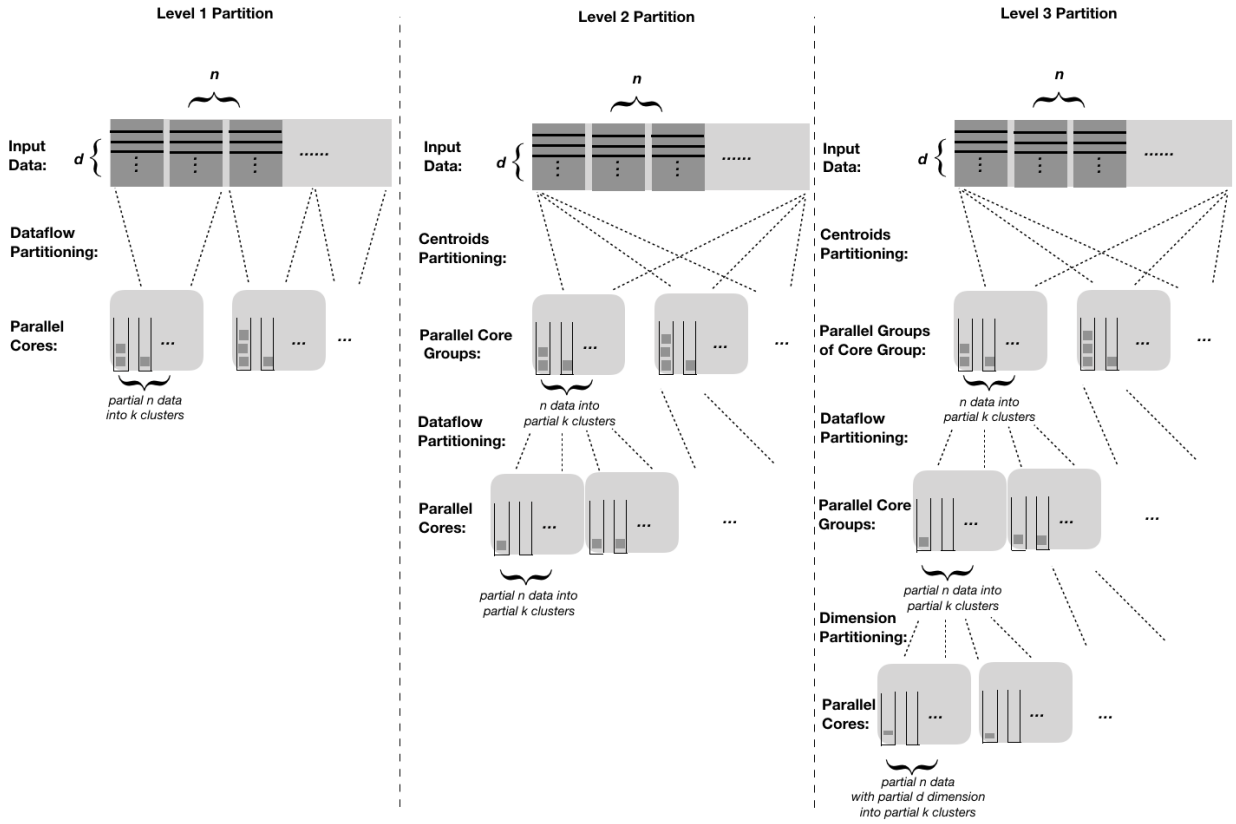


Figure 5.1: Three-level *k-means* design for data partition on parallel architectures

Algorithm 2 Basic Parallel *k-means*

INPUT: Input dataset $\mathcal{X} = \{x_i | x_i \in R^d, i \in [1, n]\}$, and initial centroid set $\mathcal{C} = \{c_j | c_j \in R^d, j \in [1, k]\}$
$P_l \xleftarrow{load} \mathcal{C},\ l \in \{1 \dots m\}$
repeat
  // Parallel execution on all CPEs:
  for $l = 1$ *to* m **do**
    Init a local centroids set $\mathcal{C}^l = \{c_j^l | c_j^l = \mathbf{0}, j \in [1, k]\}$
    Init a local counter $count^l = \{count_j^l | count_j^l = 0, j \in [1, k]\}$
    for $i = (1 + (l-1) * \frac{n}{m})$ *to* $(l * \frac{n}{m})$ **do**
      $P_l \xleftarrow{load} x_i$
      $a(i) = arg\ min_{j \in \{1 \dots k\}} dis(x_i, c_j)$
      $c_{a(i)}^l = c_{a(i)}^l + x_i$
      $count_{a(i)}^l = count_{a(i)}^l + 1$
    for $j = 1$ *to* k **do**
      AllReduce c_j^l and $count_j^l$
      $c_j^l = \frac{c_j^l}{count_j^l}$
until $\mathcal{C}^l == \mathcal{C}$
OUTPUT: $\mathcal{C}$

Level 1 - DataFlow Partition

In the simple case, the first step is processing, *Assign*, on each CPE in parallel while using multi-CPE collaboration to implement the second step, *Update*. The pseudo code of this case is shown in Algorithm 2.

The *Assign* step is implemented similarly to the traditional parallel *k-means* algorithm – (1.1) and (1.2) as above. Given n samples, they are partitioned into multiple CPEs. Each CPE (P_l) firstly reads one sample x_i and finds the minimum distances *dis* from the sample to all centroids c_j to obtain $a(i)$. Then two variables, $c^l_{a(i)}$ and $count^l_{a(i)}$, are accumulated for each cluster centroid c_j according to $a(i)$, shown in line 11 and 12. The first variable stores the vector sum of all the samples assigned to c_j, notated as $c^l_{a(i)}$. The second variable counts the total number of samples assigned to c_j, notated as $count^l_{a(i)}$.

In the *Update* step, the c^l_j and $count^l_j$ are first accumulated for all CPEs by performing two synchronisation operations. So that all CPEs can obtain the assignment results of the whole input dataset. *register communication* [117] is applied to implement intra-CG AllReduce operation and MPI_AllReduce is used for inter-CG AllReduce. After the accumulation, the *Update* step is performed to calculate new centroids, as shown in line 15.

Considering a one-CG task, constraints are analysed on scalability in terms of memory limitation of each CPE. Based on the steps above, one CPE has to accommodate at least one sample x_i, all cluster centroids $\mathcal{C}$, k centroids' accumulated vector sum $\mathcal{C}^l$ and k centroids' counters $count^l$. Considering that each CPE has a limited size of LDM, then there is a constraint ($\mathbf{C}_1$) below:

$$\mathbf{C}_1: \qquad d(1+k+k)+k \leq LDM$$

Since both the number of centroids k and the dimension d for each sample x_i should at least be 1, there are two more boundary constraints ($\mathbf{C}_2$) and ($\mathbf{C}_3$) below, separately:

$$\mathbf{C}_2: \qquad 3d+1 \leq LDM$$

$$\mathbf{C}_3: \qquad 3k+1 \leq LDM$$

As for performance, note that the *Assign* step of computing $a(i)$ for each sample x_i is completed fully in parallel on the m CPEs. Given the bandwidth of multi-CPE architecture to be B, the DMA time of reading data from main

memory can be simply formalized as:

$$\mathbf{T}_{read}: \quad (\frac{n*d}{m} + k*d)/B$$

Theoretically, a linear speedup for computing time against the serial implementation can be obtained for the *Assign* step if $m = n$ CPEs can be applied in total.

Two synchronisation operations are the bottleneck process in the *Update* step. The *register communication* [117] technique for internal multi-CPE communication guarantees a high-performance with a normally 3x to 4x speedup than other on-chip and Internet communication techniques (such as DMA and MPI) for this bottleneck process (referring to the experimental configuration section for detailed quantitative values). Given the bandwidth of *register communication* to be R, the time for the AllReduce process can be formalised as:

$$\mathbf{T}_{comm}: \quad \frac{n}{m}((1+k)*d)/R$$

Level 2 - DataFlow and Centroids Partition

Multiple (up to 64) CPEs are used in one CG to partition the set of centroids to scale the number of k for cluster centroids $\mathcal{C}$. The number of CPEs grouped to partition the centroids is denoted by m_{group}. For illustration, l' is used to index the CPE groups {P}. Then there is:

$$\{P\}_{l'} := \{P_l\},\ l \in (1 + (l'-1)*m_{group}, l'*m_{group})$$

The pseudo code of this case is shown in Algorithm 3. A new sub-step beyond previous case is to partition k centroids on m_{group} CPEs as shown in line 2. Different with the *Assign* step in above case, each data sample x_i is partitioned in each CPE group as shown in line 8. After that, similar to (1.2), all P_l in each $\{P\}_{l'}$ can still compute a partial value of $a(i)$ (named as $a(i)'$) fully in parallel without communication. Note that the domain of j in line 11 is only a subset of $(1, \ldots, k)$ as presented above in line 2, so one more step is needed to achieve data communication between CPEs in each CPE group to obtain the final a(i) as shown in line 10.

The *Update* step is similar to previous case. The scheduler just views one CPE group as one basic computing unit, which does what a CPE did in the previous case. Each CPE only computes values of subset of centroids $\mathcal{C}$ and

Algorithm 3 Parallel *k-means* for *k*-scale

INPUT: Input dataset $\mathcal{X} = \{x_i | x_i \in R^d, i \in [1, n]\}$, and initial centroid set $\mathcal{C} = \{c_j | c_j \in R^d, j \in [1, k]\}$
$P_l \xleftarrow{load} c_j \;\; j \in (1 + mod(\frac{l-1}{m_{group}}) * \frac{k}{m_{group}}, \;\; (mod(\frac{l-1}{m_{group}}) + 1) * \frac{k}{m_{group}})$
repeat
    // Parallel execution on each CPE group $\{P\}_{l'}$:
    for $l' = 1$ *to* $\frac{m}{m_{group}}$ **do**
        Init a local centroids set $\mathcal{C}^{l'}$ and counter $count^{l'}$
        for i $= (1 + (l' - 1)\frac{n * m_{group}}{m})$ *to* $(l'\frac{n * m_{group}}{m})$ **do**
            $\{P\}_{l'} \xleftarrow{load} x_i$
            $a(i)' = arg\ min_j\ dis(x_i, c_j)$
            $a(i) = min.\ a(i)'$
            $c^{l'}_{a(i)} = c^{l'}_{a(i)} + x_i$
            $count^{l'}_{a(i)} = count^{l'}_{a(i)} + 1$
        for j $= (1 + mod(\frac{l-1}{m_{group}}) * \frac{k}{m_{group}})$ *to* $((mod(\frac{l-1}{m_{group}}) + 1) * \frac{k}{m_{group}})$ **do**
            AllReduce $c^{l'}_j$ and $count^{l'}_j$
            $c^{l'}_j = \frac{c^{l'}_j}{count^{l'}_j}$
until $\cup\, \mathcal{C}^{l'} == \mathcal{C}$
OUTPUT: $\mathcal{C}$

does not need further communications in this step as it only needs to store this subset.

Concerning the scalability analysis of k in this case, the number of original k centroids distributed in m_{group} CPEs leads to a easier constraint of k against the ($\mathbf{C}_3$) above:

$$\mathbf{C'}_3 : \ 3k + 1 \leq m_{group} \ * LDM \ (m_{group} \leq 64)$$

Based on this, the ($\mathbf{C}_1$) can also be easily scaled as follow:

$$\mathbf{C'}_1 : \ d(1 + k + k) + k \leq m_{group} \ * LDM \ (m_{group} \leq 64)$$

Note that the scheduler still need to accommodate at least one d-dimensional sample in one CPE, so the ($\mathbf{C}_2$) should be kept as before: $\mathbf{C'}_2 := \mathbf{C}_2$

As for performance, since m_{group} CPEs in one group should read the same sample simultaneously, the processors need more time to read the input data samples than the first case, but only partial cluster centroids need to be read by each CPE:

$$\mathbf{T'}_{read} : \quad (\frac{n * d * m_{group}}{m} + \frac{k}{m_{group}} * d)/B$$

As for the data communication needed, there is one more bottleneck process (line 12) than before. Comparing against the above cases, multiple CPE groups can be allocated in different processors. Those communications are done through MPI which is much slower than internal processor multi-CPEs *register communication*. Given the bandwidth of network communication through MPI to be M, there is:

$$\mathbf{T'}_{comm} : \frac{k}{m_{group}}/R + \frac{n * m_{group}}{m}((1 + k) * d))/M$$

Level 3 - DataFlow and Centroids and Dimensions Partition

One d-dimensional sample is stored and partitioned by one CG using up-to 64 CPEs to scale the number of dimension d for each sample x_i. The pseudo code of this case is shown in Algorithm 4.

Recall u is used to index the data dimension: $u \in (1 \ldots d)$; Now l'' is used to index the CGs and m'_{group} to denote the number of CGs grouped together

Algorithm 4 Parallel *k-means* for k-scale and d-scale

INPUT: Input dataset $\mathcal{X} = \{x_i | x_i \in R^d, i \in [1, n]\}$, and initial centroid set $\mathcal{C} = \{c_j | c_j \in R^d, j \in [1, k]\}$
$CG_{l''} \xleftarrow{load} c_j^d,\ l'' \in \{1 \dots \frac{m}{64}\}, j \in (1+mod(\frac{l''-1}{m'_{group}}) * \frac{k}{m'_{group}},\ (mod(\frac{l''-1}{m'_{group}}) + 1) * \frac{k}{m'_{group}})$
repeat
  // Parallel execution on each CG group $\{CG\}_{l''}$:
  for $l'' = 1$ *to* $\frac{m}{64}$ **do**
    Init a local centroids set $\mathcal{C}^{l''}$ and counter $count^{l''}$
    for i $= (1 + (l''-1)\frac{n*m'_{group}}{m})$ *to* $(l''\frac{n*m'_{group}}{m})$ **do**
      for u $= (1 + mod(\frac{l-1}{64}) * \frac{d}{64}$ *to* $(mod(\frac{l-1}{64}) + 1) * \frac{d}{64})$ **do**
        $CG_{l''} \leftarrow x_i\ (P_l \leftarrow x_i^u)$
      $a(i)' = arg\ min_j\ dis(x_i, c_j)$
      $a(i) = min.\ a(i)'$
      $c_{a(i)}^{l''} = c_{a(i)}^{l''} + x_i$
      $count_{a(i)}^{l''} = count_{a(i)}^{l''} + 1$
    for j $= (1 + mod(\frac{l''-1}{m'_{group}}) * \frac{k}{m'_{group}})$ *to* $((mod(\frac{l''-1}{m'_{group}}) + 1) * \frac{k}{m'_{group}})$ **do**
      AllReduce $c_j^{l''}$ and $count_j^{l''}$
      $c_j^{l''} = \frac{c_j^{l''}}{count_j^{l''}}$
until $\cup\, \mathcal{C}^{l''} == \mathcal{C}$
OUTPUT: $\mathcal{C}$

to partition k centroids. Consider that m is represented the total number of applied CPEs and each CG contains 64 CPEs, then there are

$$l'' \in (1, \ldots, \frac{m}{64}), m'_{group} \leq \frac{m}{64}$$

$$CG_{l''} := \{P_l\},\ l \in (1 + 64(l'' - 1),\ 64l'')$$

An updated step is applied to partition k centroids on multiple CGs against the previous case as shown in line 2. Line 9 shows the step to partition each d-dimensional sample x_i^d on 64 CPEs in one CG. Similar to the above case, all $CG_{l''}$ in each CG group compute the partial value $a(i)'$ fully in parallel and then communicate to obtain the final $a(i)$. Multi-CG communication in multiple many-core processors (nodes) is implemented through MPI interface. The *Update* step is also similar to the previous case. Now one CG is viewed as one basic computing unit which conducts what one CPE did before and what a CG group does is the same as what a CPE group did before.

In this case, each CG with 64 CPEs accommodates one d-dimensional sample x_i. Then the previous ($\mathbf{C}_2$) can be scaled as follow:

$$\mathbf{C''}_2 :\ 3d + 1 \leq 64 * LDM$$

Consider that there are totally m'_{group} CGs to accommodate k centroids in this case, then ($\mathbf{C}_3$) will scale as follow:

$$\mathbf{C''}_3 :\ 3k + 1 \leq m'_{group} * 64 * LDM$$

Note that the domain of m'_{group} seems limited by the total number of CPEs applied, m. But in fact, this number can be further scaled as this work targets supercomputers with tens of millions of cores. Finally, ($\mathbf{C}_1$) will scale as follow:

$$\mathbf{C''}_1 :\ d(1 + k + k) + k \leq 64 * m'_{group} * LDM$$

which is equal to:

$$\mathbf{C''}_1 :\ d(1 + k + k) + k \leq m * LDM$$

$\mathbf{C''}_1$ is the main contribution over other state-of-the-art work [9]: the total amount of $d * k$ is not limited by a single or shared memory size any more. It is fully scalable by the total number of processors applied (m). In a modern supercomputer, this value can be large-scaled to tens of millions when needed.

Considering performance, note that m'_{group} CGs (64 CPEs in each) in one group should read the same sample simultaneously. In another aspect, each CPE only needs to read a partial of the given d-dimension of original data sample together with a partial of k centroids similarly as before, then a similar reading time can be obtained:

$$\mathbf{T''}_{read}: \quad (\frac{n*d*m'_{group}}{m} + \frac{k}{m'_{group}} * \frac{d}{64})/B \tag{5.1}$$

Comparing with the above cases, multiple CGs in CG groups allocated in different many-core processors need communication to update centroids through MPI. Given the bandwidth of network communication through MPI to be M, the cost between multiple CG groups can be formalised as:

$$\mathbf{T''}_{read}: \quad (\frac{n*d*m'_{group}}{m} + \frac{k}{m'_{group}} * \frac{d}{64})/B \tag{5.2}$$

The network architecture of Sunway TaihuLight is a two-level fat tree. 256 computing nodes are connected via a customized inter-connection board, forming a *super-node*. All super-nodes are connected with a central routing server. The intra super-node communication is more efficient than the inter super-node communication. Therefore, in order to improve the overall communication efficiency of the proposed design, any CG group should be located within a super-node if possible.

Automatic Multi-level Partitioning

An automatic method to partition dataflow into 3 levels based on the targeting k values. This method is mainly guided by the scalability of each level of data partitioning. Based on the limitations presented in formulations $(\mathbf{C}_1, \mathbf{C'}_1, \mathbf{C''}_1)$ above, it is easy to compute the range of possible k values for each level: $k \leq \frac{LDM-d}{1+2d}$ for level-1, $k \leq \frac{64LDM-d}{1+2d}$ for level-2 and $k \leq \frac{m*LDM-d}{1+2d}$ for level-3. By concatenating the ranges, it results in the automatic 3-stage roofline model to guide the data partitioning as shown in figure 5.2.

5.3.2 Hyper-Parameter Determination

A critical parameter of the k-means algorithm, the number of clustering (k) need to be predetermined for typical experiments. Zhang et al. claim in their

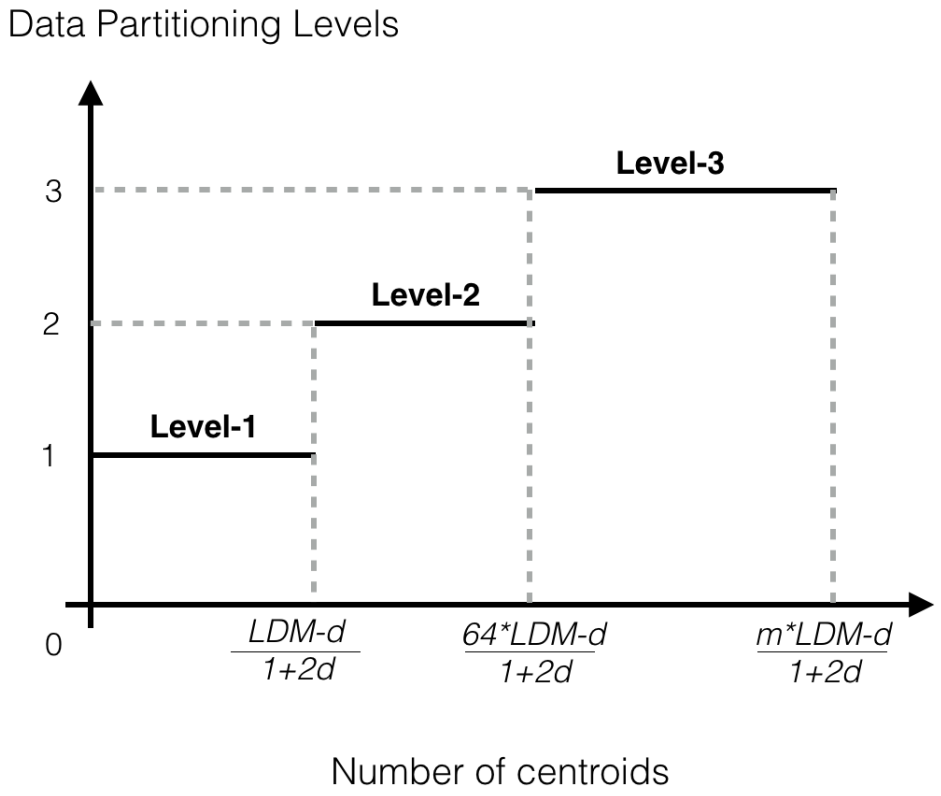


Figure 5.2: The roofline model for automatic data partitioning. SupCOLAB changes its partitioning levels based on the complexity of the input. X-axis represents the complexity of input indicated by the number of targeting centroids and the value of Y-axis corresponds to the 3 different partitioning levels

review paper[116] that how to define this value is a critical question for the community, and an inappropriate decision wwill yield poor quality clustering results.

Shi, et al. [91] proposed a basic method by gradually increasing the possible number of clusters and used the result when the distortion of solutions between current k and k-1 is less than a static predefined threshold. Chen, et al. [23] recently presented a method without any predefined threshold. It generates a formula by computing the difference between sum of distance inter and intra clusters.

This approach was found not to work in large-scale cases as it keeps monotonically increasing when the k is greater than 2.

The notion of cluster radius $r(k)$ is introduced into *k-means* clustering to solve this problem. $r(k)$ is defined to be the smallest non-negative real number such that the sample set $\mathcal{X}^d$ can be covered by k closed balls centred at sample points with radius $r(k)$. Expressed as:

$$r(k) = \inf\{t : \exists y_1, \ldots, y_k \text{ in } R^d, \mathcal{X}^d \subseteq \bigcup_{1 \leq s \leq k} B(y_s, t)\},$$

where $B(y_s, t)$ stands for the Euclidean Closed Ball centred at y_s with radius t. For instance, when $k = n$ the number of samples, there is $r(n) = 0$. It is easy to see that $r(k)$ is non-increasing with respect to k. Radius has been widely used in clustering problems, such as approximating clustering [4] and incremental clustering [22], but not on *k-means*, because it is impossible to compute and measure all possible radius values on large-scale datasets. For n samples clustering into k centroids, there will be $O(n^k)$ possible solutions.

With the support of modern supercomputers with efficient parallel processing techniques, an empirical method can be applied. It uses a minimum radius from a random selection of solutions with k centroids, named $r'(k)$ to represent the $r(k)$. With the increasing of k, the accuracy of $r'(k)$ will decrease. The $r'(k)$ will even increase at some point when it is too difficult to show a reasonable representation of $r(k)$ by $r'(k)$ from a limited selection of solutions. This also indicates that to keep increasing the targeted centroids (k) beyond this points becomes meaningless as it cannot easily distinguish the difference between different clusters. The idea is to determine the best k by measuring the change of $r'(k)$ with respect to $r(k)$. If $r'(k)$ does not keep the same trend of $r(k)$, the current k would be regarded as a reasonable

choice choice. More formally, let

$$\Delta r'(k) = r'(k) - r'(k+1),$$

then the optimal k is taken the first time this function $\Delta r'$ increasing.

5.4 SupCOLAB Scheduling Algorithm

After partitioning the large-scale dataflow, it becomes much easier to design the heterogeneity-aware scheduler on the particular supercomputer compared with other general scheduler design targeting workload without preknowledge. With sufficient knowledge of the parallel threads, the scheduler doesn't need to rely on any additional ranking heuristic or runtime prediction model to give intelligent decisions, but a simple greedy approach with a cost function of dividing the resources and assigning corresponding threads should be work.

This section presents the SupCOLAB scheduling algorithm targeting the customised design large-scale k-means workloads on Sunway Taihulight supercomputer.

Algorithm 5 SupCOLAB Scheduling Algorithm

1: **INPUT:** Cost function $T''(k, m, m')$, number of CPEs p, number of CPEs per CG q, number of points n
2: **for** i=1 to $n-1$ **do**
3: $m_i = q$; $m'_i = 0$
4: remProc $= p - (n-1) * q$
5: **while** remProc > 0 **do**
6: $i = \text{argmin}_{j=1}^{n-1} T''(j, m_j, m'_j)$
7: $m_i = m_i + q$; remProc = remProc $-q$
8: **for** i=1 to k **do**
9: $m'_i = \text{argmin}_{j=1}^{m_i, j|m_i} T''(i, m_i, j)$
10: **OUTPUT:** $\{m_1, \ldots, m_{n-1}, m'_1, \ldots, m'_{n-1}\}$

Recall n is used to represent the total number of data samples. The maximum meaningful number of targeting centroids is $n/2$ based on the Dirichlet's drawer principle [1]. Dividing the resources of a supercomputer between the $n/2$ instances of the k-means algorithm is viewed as the main

scheduling problem, where the scheduler needs to assign $n/2$ heterogeneous tasks on a given set of resources. The tasks are heterogeneous because, for different k, k-means algorithm will do different partitioning of the data (see the previous section) which yields different degree of parallelism and different reading, computation and communication costs. Therefore, dividing the resources uniformly between the instances of the algorithm (tasks) will be sub-optimal. Furthermore, it is not possible to statically compute the precise cost of executing one instance of the algorithm on a given set of resources because, in addition to the reading (equation 5.1) and communication (equation 5.2) time that can easily be estimated, there is also a computation time that depends on the number of iteration for a particular value of k and a particular input, and this number cannot be computed statically.

The SupCOLAB algorithm focuses on resource allocation and scheduling for the most complicated large-scale scenario, the level-3 partitioning. The cost function applied here, $T''(k, m, m'_{group})$, is an estimation of the cost of executing an instance of k-means on m CPEs and m'_{group} CPE groups for each centroid. For brevity, m'_{group} is annotated with m'. The scheduling problem can then be seen as the optimisation problem of finding the minimum of the function:

$$A(m_1, \ldots, m_{\frac{n}{2}}, m'_1, \ldots m'_{n-1}) = \sum_{i=1}^{\frac{n}{2}} T''(i, m_i, m'_i)$$

with the following constraints: $1 \leq m_i \leq p$ (for $i \in \{1, \ldots, \frac{n}{2}\}$), $\sum_{i=1}^{\frac{n}{2}} m_i \leq p, 0 \leq m'_i \leq \frac{p}{q}$ (for $i \in \{1, \ldots, \frac{n}{2}\}$), $\sum_{i=1}^{\frac{n}{2}} m'_i = \frac{p}{q}, m'_i | m_i$; (for $i \in \{1, \ldots, \frac{n}{2}\}$ where p is the total number of CPEs and q is the number of CPEs per group (64 in this case). Due to a way in which the data partitioning is done, it is required that each m to cover at least one core group, i.e. to be a multiple of 64[1]. Then the cost function can be formalized as

$$T''(k, m, m') = T''_{read}(k, m, m') + T''_{comm}(k, m, m')$$

where T''_{read} and T''_{comm} are given in the equations 5.1 and 5.2.

The SupCOLAB algorithm is given in Algorithm 5, which is based on a greedy approach. Note that, in theory, for level-3 scheduling the scheduler

[1]In other words, the scheduler is really allocating core groups to tasks, rather than just individual CPEs

would need to consider allocation of individual CPEs (level-1), CGs (level-2) and CG groups (level-3) to the instances of the k-means algorithm. However, an easy way to present the approach and simplify the problem is by assuming that no CG will share its resources between different instances of the algorithm. Therefore, the basic unit of allocation will be CG. The parameters of the algorithm are cost function, T'', number of available CPE groups (CGs), p, number of CPEs per CG, q, and the number of points n.

Initially, one CG with zero CG groups is allocated to each of the $n/2$ instances of the k-means algorithm (lines 2–4). Then, in successive iterations, one more CG is added to the instance which has the highest cost (therefore reducing its cost), until all of the CGs are allocated (lines 6–9). This, effectivelly, gives the assignment of $m_1, m_2, \ldots, m_{n-1}$ (m_i will be the number of CGs allocated to the instance i multiplied by q (64 in this case). Once the number of CGs for instances has been decided, these CGs are divided into CG groups and used to find, for each instance, the grouping that minimised T'' (line 11). This gives the assignment of $m'_1, m'_2, \ldots, m'_k$.

5.5 Experimental Setup

The datasets applied in experiments come from well-known benchmark suites including UCI Machine Learning Repository[81] and ImgNet [54], as discussed in the technical background chapter.

The experiments have been conducted to demonstrate scalability, high performance and flexibility by increasing the number of centroids k and number of dimensions d on multiple datasets with vary data size n. The three-level designs are tested targeting different benchmarks. Different hardware setup on Sunway Taihulight Supercomputer will be provided for testing different scalable levels:

- *Level 1* - One SW26010 many-core processor is used, which contains 256 64-bit RISC CPEs running at 1.45 GHz, grouped in 4 CGs in total. 64 KB LDM buffer is associated with each CPE and 32 GB DDR3 memory is shared for the 4 CGs. The theoretical memory bandwidth for register communication is 46.4 GB/s and for DMA is 32 GB/s.

- *Level 2* - Up-to 256 SW26010 many-core processors are used, which contains 1,024 CGs in total. The bidirectional peak bandwidth of the network between multiple processors is 16 GB/s.

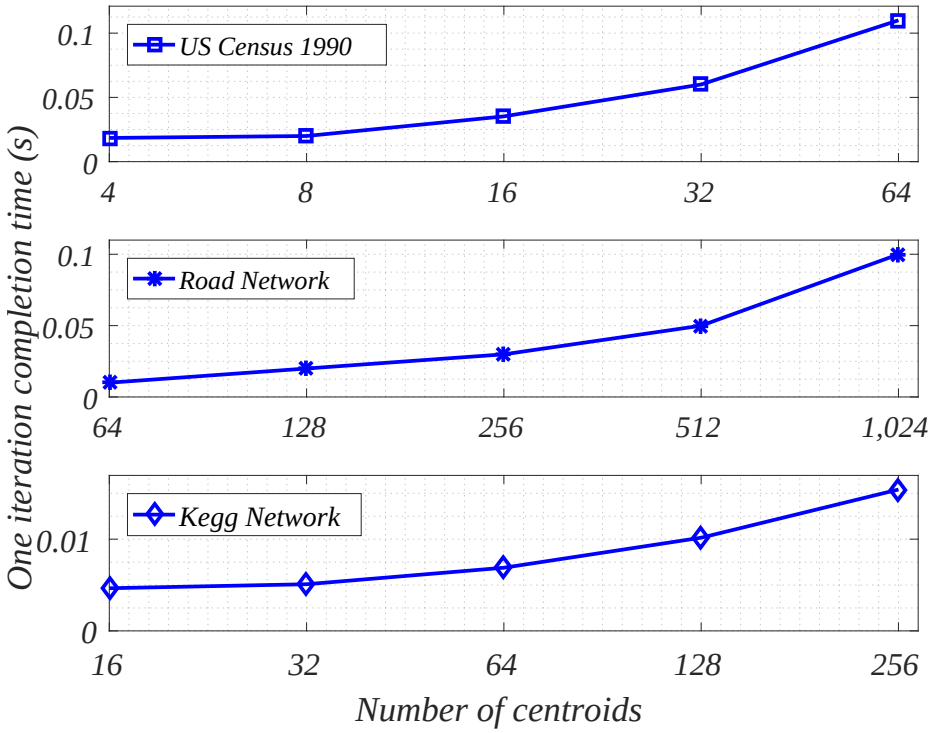


Figure 5.3: Level 1 - dataflow partition using UCI datasets

- *Level 3* - Up-to 4,096 SW26010 many-core processors are used, which contains 16,384 CGs in total.

The main performance metric concerned here is *one iteration completion time*. Note that the total number of iterations needed and the quality of the solution (precision) are not considered in the experiments as this work does not relate to the optimisation of the underlining Lloyd algorithm or the solution of *k-means* algorithm.

5.6 Results

This section presents the experimental results. The first subsection demonstrates the high performance and large-scalability of the proposed workload generator with hierarchical data partitioning targeting the customised resources on the given supercomputer. The second subsection shows how the whole heterogeneity-aware supercomputer oriented approach, equipped with

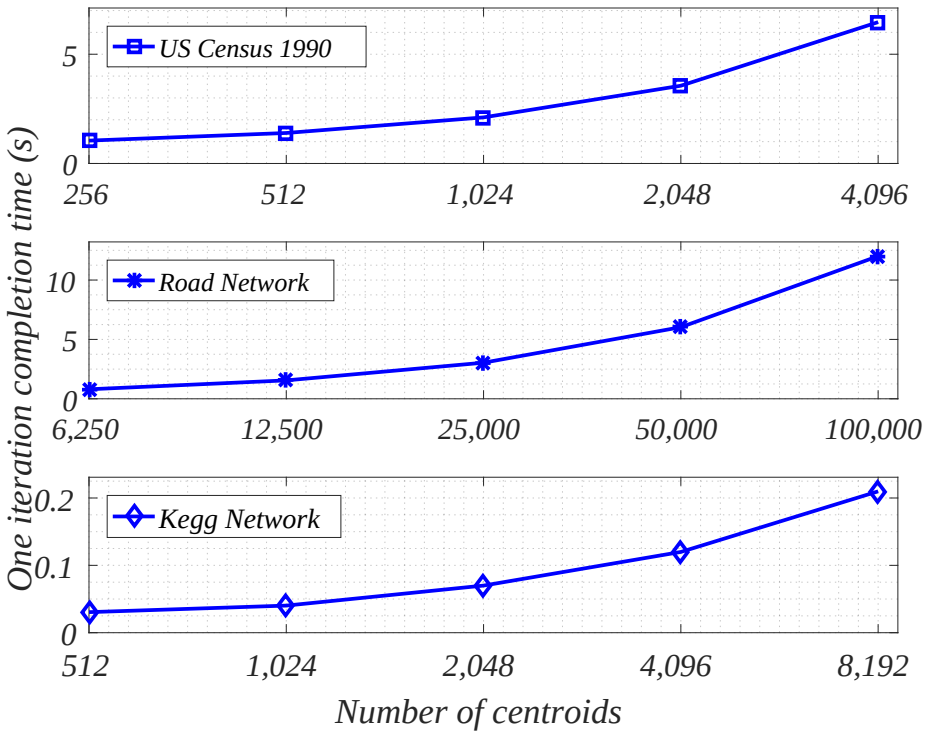


Figure 5.4: Level 2 - dataflow and centroids partition using UCI datasets

both the data partitioner and scheduler, can be applied to efficiently solve real applications with massive data.

5.6.1 Results of Multi-level Data Partitioning

Level 1 - dataflow partition

The *Level 1* (n-partition) parallel design is applied to three UCI datasets (*US Census 1990, Road Network, Kegg Network*) with their original sizes (n = 2,458,285, 434,874 and 65,554 separately) and data dimensions (d = 68, 4 and 28) for different numbers of targeting centroids (k). The purpose of these experiments is to demonstrate the efficiency and flexibility of this approach on datasets with relatively low size, dimensions and centroid values. Figure 5.3 shows the *one iteration completion time* for those datasets over increasing number of clusters, k. As the number of k increases, the completion time on this approach grows.

Level 2 - dataflow and centroids partition

The level 2 (nk-partition) parallel design is applied to same three UCI datasets as above, but for a large range of target centroids (k). The purpose of these experiments is to demonstrate the efficiency and flexibility of the proposed approaches on datasets with large-scale target centroids (less than 100,000). Figure 5.4 shows the *one iteration completion time* of the three datasets of increasing number of clusters, k. As the number of k increases, the completion time from this approach grows linearly. In conclusion, this approach works well when one dimension is varied up to the limits previously published.

Level 3 - dataflow, centroids and dimensions partition

The *Level 3* (nkd-partition) parallel design is applied to a subset of ImgNet datasets (*ILSVRC2012*) with its original size (n = 1,265,723). The results are presented with varying number of target centroids (k) and data dimension size (d) with an extremely large domain. The scalability is also tested by varying the number of computational nodes. The purpose of these experiments is to demonstrate the high performance and scalability of the proposed approaches on datasets with large size, extremely high dimensions and target centroids. Figure 5.5 shows the completion time of the dataset of increasing number of clusters, k = 128, 256, 512, 1024 and 2,048 with increasing

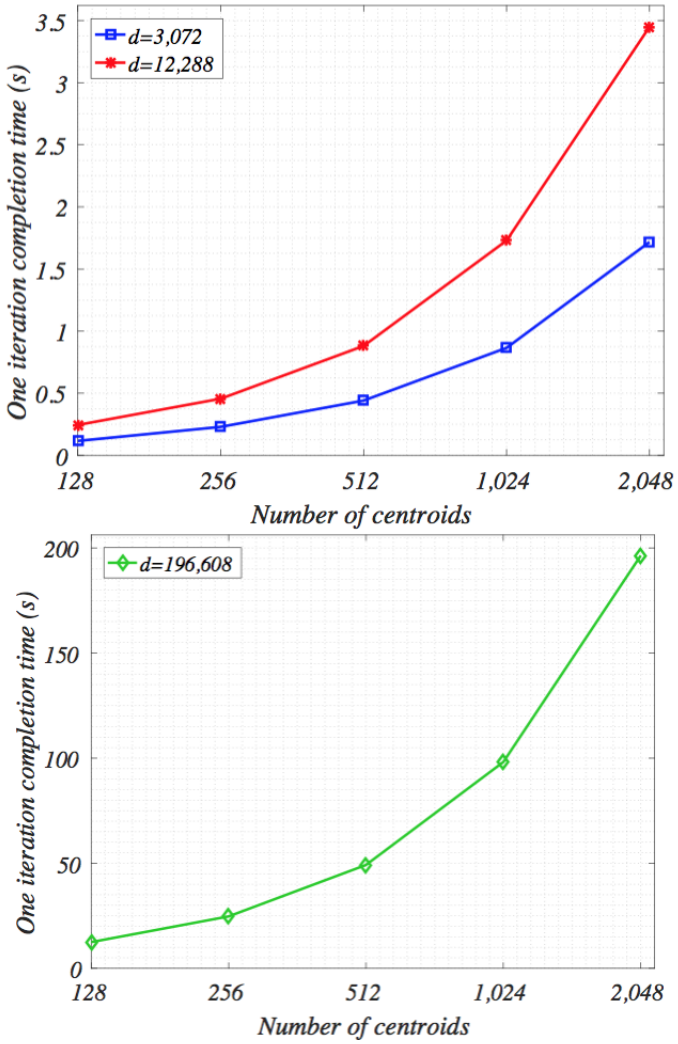


Figure 5.5: Level 3 - dataflow, centroids and data-sample partition using a subset of ImgNet datasets (*ILSVRC2012*)

number of dimensions, d = 3,072 (32*32*3), 12,288 (64*64*3) and 196,608 (256*256*3).

Two more cases are tested to further investigate the scalability of the proposed approach by either further scaling centroids by certain number of data dimensions (d = 3,072) and number of nodes ($nodes$ = 128) or further scaling nodes applied by certain number of data dimensions (d = 196,608) and number of centroids (k = 2,000). The results of those two tests are shown in Figure 5.5.

As both k and d increase, the completion time from the proposed approach continues to scale well, demonstrating the claimed high performance and scalability.

Comparison of partition levels

In this section, the Level 2 and Level 3 approaches are experimentally compared.

Figure 5.7 shows how *one iteration completion time* grows as the number of dimensions increases. The Level 2 approach outperforms Level 3 when the number of dimensions is relatively small. However, the Level 3 approach scales significantly better with growing dimensionality, outperforming Level 2 for all d greater than 2560. The Level 2 approach cannot run with d greater than 4096 in this scenario due to memory constraints. However, it is clear that, even if this problem were solved, the poor scaling would still limit this approach. The completion time for Level 2 falls twice unexpectedly between 1536 and 2048, and between 2560 and 3072. This is due to the crossing of communication boundaries in the architecture of the supercomputer - DMA can fully use the bandwidth if and only if the reading granularity is greater than 256 bytes and is an integral multiple of 128. Otherwise the bandwidth resources will be wasted.

Figure 5.8 shows how the *one iteration completion time* grows as the number of centroids, k increases. Since the number of d is fixed at 4096, the Level 3 approach actually outperforms Level 2, with the gap increasing as k increases. This scaling trend is replicated at lower levels of d too, though Level 2 initially outperforming Level 3 at lower values of k.

Figure 5.9 shows how both Level 2 and Level 3 scale across an increasing number of computation nodes. Level 3 clearly outperforms Level 2 in all scenarios. The values of k and d are fixed, as described in the graph caption, at levels which Level 2 can operate. The performance gap narrows as more

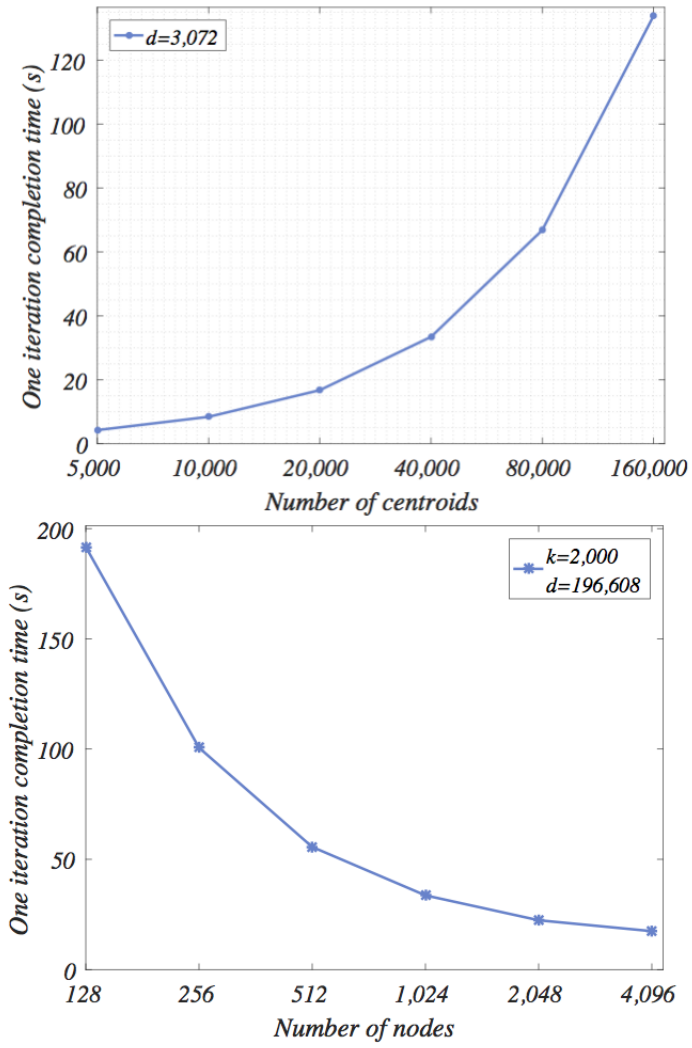


Figure 5.6: Level 3 - large-scale on centroids and nodes using a subset of ImgNet datasets (*ILSVRC2012*)

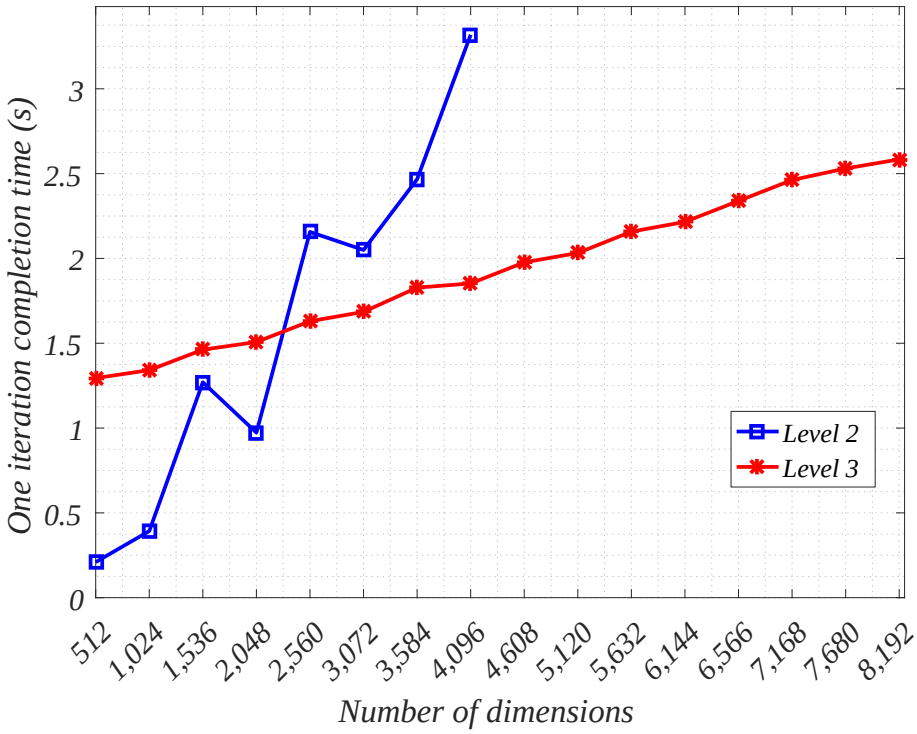


Figure 5.7: Comparison: varying d with 2,000 centroids and 1,265,723 data samples tested on 128 nodes using a subset of ImgNet datasets (*ILSVRC2012*)

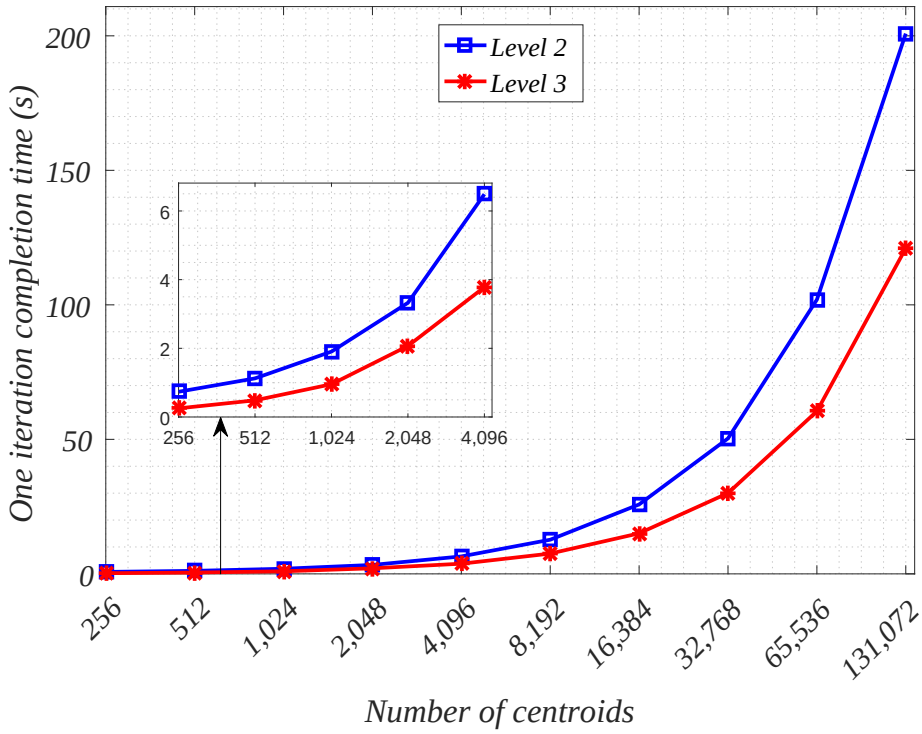


Figure 5.8: Comparison test: varying k with 4,096 Euclidean dimensions and 1,265,723 data samples tested on 128 nodes using a subset of ImgNet datasets (*ILSVRC2012*)

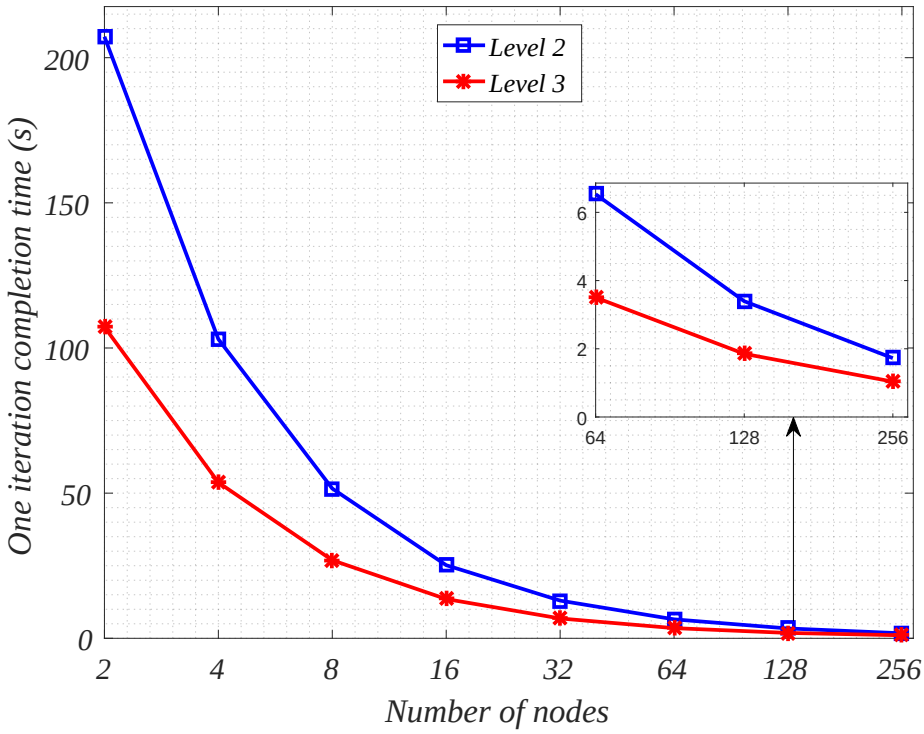


Figure 5.9: Comparison test: varying number of nodes used with a fixed 4,096 Euclidean dimension, 2,000 centroids and 1,265,723 data samples using a subset of ImgNet datasets (*ILSVRC2012*)

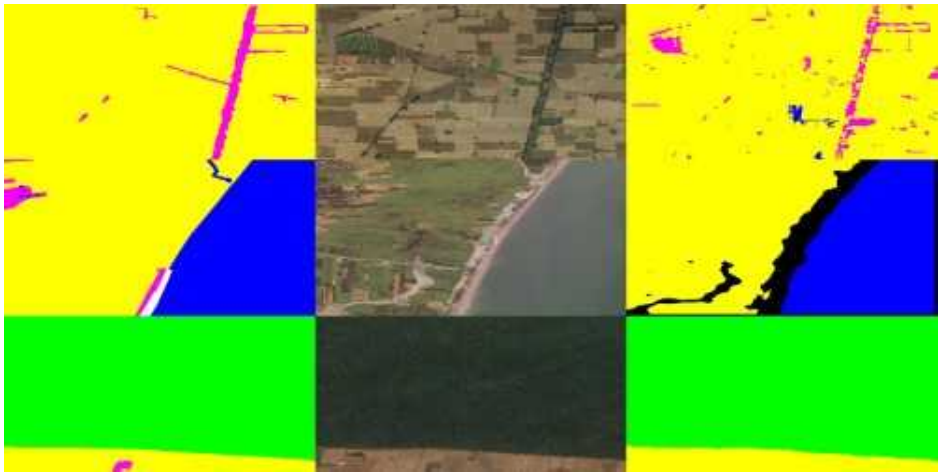


Figure 5.10: Remote Sensing Image Classification. Left: the result from baseline approach provided by [33]. Middle: the corresponding original image. Right: the proposed classification result. Different colors are applied to identify different region classes as used in [33].

nodes are added, but remains significant. Clearly the exact performance numbers will vary with other values k and d, as can be inferred from other results, but the main conclusion here is that Level 3 generally scales well.

5.6.2 Results of Real Applications

Land Cover Classification

This is a popular remote sensing problem, requiring unsupervised methods to handle high numbers of unlabeled remote sensing images [70]. *K-means* algorithm has already been used for regional land cover classification with small number of targeted classes. For example, Figure 5.10 shows the proposed result of classifying a remote sensing image (from a public dataset called Deep Globe 2018 [33]) into 7 classes, representing the urban, the agriculture, the rangeland, the forest, the water, the barren and unknown. There are 803 images in the Deep Globe 2018 dataset, and each image has about $2k \times 2k$ pixels. The resolution of the image is 50cm/pixel. In this problem definition, one image is been processed, where n is 5838480, k is 7 and d is 4096, which can be done with 400 SW26010 many-core processors. The Level 3 design can process the clustering dataset efficiently.

In recent years, high-resolution remote sensing images have become more common in land cover classification problems. The problem definition on high-resolution images is more complex as the classification sample can be a block of pixels instead of one pixel, which means the d can be even larger.

Gene Expression Data Classification

Genomic information from gene expression data has been widely used and already benefited on improving clinical decision and molecular profiling based patient stratification. Clustering methods, as well as their corresponding HPC-based solutions[103], are adopted to classify the high-dimensional gene expression sequences into some known patterns, which indicates that the number of targeted clustering centroids are determined in advance. There are still large numbers of gene expression sequences, among which the patterns are not yet discovered. Therefore, the proposed auto-clustering method can potentially help find new patterns from high-dimensional gene expression datasets.

In this work, the whole heterogeneity-aware data partitioning and scheduling approach is tested on the ONCOLOGY& Leukaemia gene expression datasets[41]. There are 4254 subjects and each subject has 54675 probesets. In this problem definition, whole dataset is clustered using the level-3 partitioning method, where n is 4254, and d is 54675. In this task, the candidate k is generated by enumerating from 2 to 2000 (up-to around $n/2$). The performance for one iteration execution time is shown in figure 5.11 and the total execution time is shown in figure 5.12. The results demonstrate good performance of the proposed approach with a linear scale on one iteration time and also shows that the proposed supercomputer-based technique can compute such a large-scale dataset for all needed iterations within 200 seconds at most.

The evaluation function is further applied to determine the optimal value of k. The results are shown in figure 5.13. It can be seen that $r'(k)$ reaches the first increasing when $k = 14$. After that, $r'(k)$ fluctuates around a certain value, which indicates that continually increasing the k values cannot further represent more patterns in the input data.

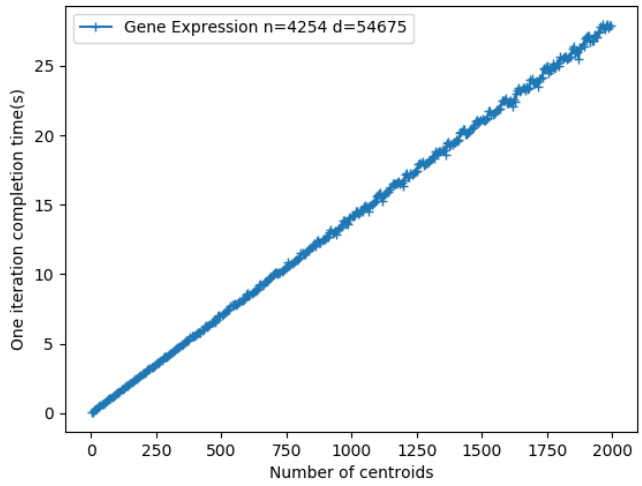


Figure 5.11: One iteration execution time for gene expression dataset ONCOLOGY and LEukemia.

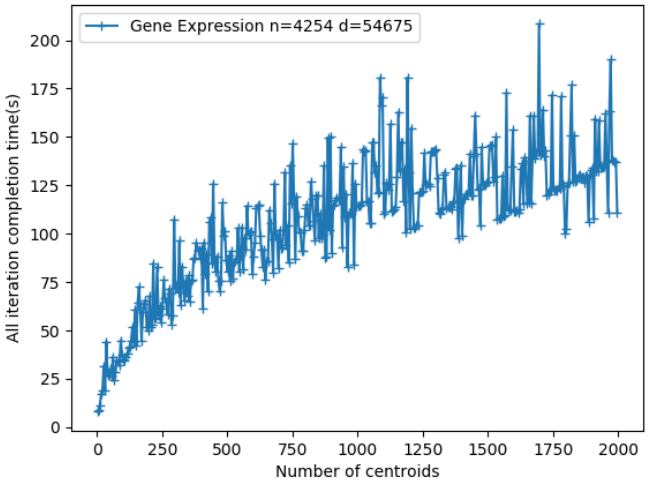


Figure 5.12: Total execution time for gene expression dataset ONCOLOGY and LEukemia.

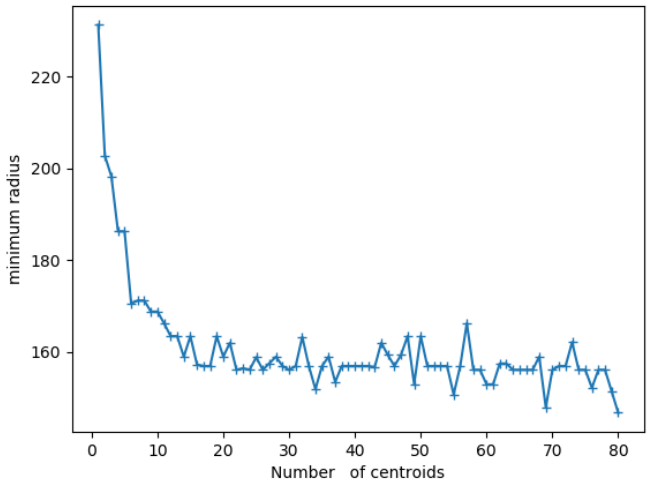


Figure 5.13: The evaluation function $r'(k)$ to determine the optimal k value.

5.7 Conclusion

In this chapter, a systematic heterogeneity-aware scheduling with data partitioning approach is conducted targeting large-scale workloads on many-core supercomputers. Instead of a general high-level theory, this work proposes a concrete study on addressing k-means based large-scale workloads on the world leading Sunway Taihuilight supercomputer running tens of millions of cores simultaneously.

Its heterogeneity-aware large-scale workload generator first provides a fully data partitioned (nkd-partition) approach for parallel *k-means* implementation to achieve scalability and high performance at large numbers of centroids and high data dimensionality simultaneously. Running on the Sunway TaihuLight supercomputer, it breaks previous limitations for high performance parallel *k-means*. Furthermore, it contains an automatic hyper-parameter determination process, by automatically generating and executing the clustering tasks with a number of candidate hyper-parameters, and then determining the optimal hyper-parameter according to a specified evaluation method. It provides a systematic approach to generate large-scale workloads on large data without pre-knowledge. Finally, it provides a simple but efficient scheduler for the supercomputer to address the customised workload

achieving high performance and large scalability.

The experimental results demonstrate not only the performance goal of the data partitioning based workload generator on machine learning benchmarks, but also how the whole heterogeneity-aware supercomputer-oriented approach can be applied to solve real world problems efficiently.

Chapter 6

AccCOLAB: A Heterogeneity-aware Scheduler for Multi-accelerator Systems

6.1 Introduction

FPGA-based accelerators have been shown to perform exceptionally well in terms of parallel performance and energy efficiency, yet the generality of this approach is a significant issue in distributed multi-accelerator environments. A key challenge for these platforms is to maintain high levels of performance on diverse workloads where task size and computational elements required vary dynamically. Existing dynamic scheduling for multi-FPGA systems rely on a classic First-In-First-Out (FIFO) approach, which this work shows to be sub-optimal. In this chapter, a topological ranking-based collaborative scheduling approach, named AccCOLAB, is presented which significantly outperforms both the existing FIFO scheduler *Orchestrator* used in the Maxeler DataFlow Engine cluster [73], but also other scheduling approaches [59, 97] targeted at other systems problems.

This work is evaluated on a Xilinx FPGA-based DataFlow Engine using multiple workloads from an industrial Reverse Time Migration (RTM) application, and a parallel genetic algorithm, both of which are real-world workloads which exhibit high levels of irregularity.

The use of multiple accelerators in additional of general purpose processors to accelerate specialised workloads in distributed computations is

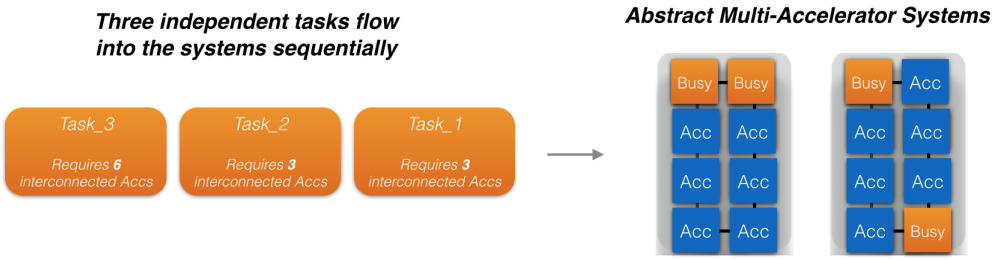


Figure 6.1: Abstract case of multi-task runtime scheduling on multi-FPGA systems. The left hand side is 3 coming tasks. The right hand side is the system states at the time of tasks arrival. FIFO approach can allocate task_1 to the first device and allocate task_2 to the first or second device, but cannot allocate task_3 to any device.

becoming increasingly important both in industry and research communities [50, 26]. New generation cloud platforms can offer specialised heterogeneous resources such as FPGAs, GPGPUs and Coarse-Grained Reconfigurable Architectures (CGRAs), which are well suited to handling challenging Quality-of-Service (QoS) requirements, and offer the opportunity for fast and efficient distributed computation.

Dynamically scheduling irregular workloads in a FPGA-based multi- accelerator system is a complex task. The scheduler need to find a way to efficiently schedule multiple irregular tasks and assign corresponding resources to hardware accelerators during runtime to find an adequate solution. This can be difficult to achieve in practice for a number of reasons: (1) Multiple tasks should be executed quickly and simultaneously to achieve a short total execution time and high throughput; (2) tasks can be instantiated on different system configurations with differing numbers of accelerators, so an ideal configuration cannot be easily determined a priori; (3) tasks require different topologies between hardware accelerators based on what communication is needed during runtime.

Motivating Example

Consider the abstract case as shown in Fig. 6.1 – a cluster containing two FPGA-based multi-accelerator devices, each of which has eight intercon-

nected accelerators[1]in a state where some accelerators are busy. Three new tasks are dynamically sent to the system, and each has different resource requirements and connectivity constraints. Using a classic First-In-First-Out (FIFO) approach, the first two tasks will be assigned to run on the first device as there is vacancy, but the third task cannot be allocated to the any device as it requires six interconnected accelerators. This means the scheduling will stall until new resources are released by the completion of the previous tasks. It is clear that current *load-balancing* based heuristics [59], which would allocate Tasks 1 and 2 to different accelerator clusters, also fail to perform in this scenario because neither of the two devices would have enough remain resources to allocate Task 3. Similarly, *Maximum-execution-time* based heuristics [97] do not necessarily help here if Task 3 would not have the longest predicated execution time although it asks for more resources – a common occurrence.

This challenge leads to the main motivation of the work presented in this chapter: to give a generalised method to design high-performance runtime scheduling methods in multi-FPGA systems targeting irregular tasks with arbitrary capacity representations. Different tasks can ask for different size and topology of accelerators: large workloads may need more accelerators to satisfy timing requirements, and tasks that are capable of exploiting multiple accelerators and specific topologies should have potential access to such resources.

This chapter proposes the AccCLOAB scheduler which efficiently schedules irregular workloads using a novel heuristic-based runtime resource allocation methodology. It uses a representation which is general enough to be used across different types of accelerator system, and is capable of representing the required resource and connectivity requirements for workloads. Leveraging this representation, a ranking-based scheduler is built.

6.2 Ranking Irregular Tasks

This section presents an original approach to representing irregular tasks targeting multi-accelerator systems. It first models these tasks and then illustrates the representation and ranking approach.

[1]In this chapter, one FPGA is referred to be one accelerator.

6.2.1 Modeling Irregular Tasks

A generalised procedure for developing a heuristic-based runtime scheduler should handle resources with arbitrary capacity and allocation representations. The primary goal is to efficiently construct heuristics for irregular multi-dimensional tasks to guide the allocation strategy. Task rankings should be *universally consistent* – that is they should have the same ranking priority during the modelling process. They should also be *easily-computable*, within polynomial-time, so not to increase the complexity. This work first defines the universal relationship between irregular multi-dimensional tasks in order to model and rank them appropriately.

Definition 1 *Given the relationship $\leq$, named less than, between tasks on a multi-FPGA system α:* $\forall A, B \in Tasks,\ \alpha(B) \mapsto \alpha(A) \Leftrightarrow A \leq B$.

Here P $\mapsto$ Q means Q must be true if P is true. $\alpha(B)$ is true iff α can serve B. So the right hand side of the equation means that for all possible states of this multi-FPGA system which can serve B, will also serve A. Then A $\leq$ B represents the observation that A asks for less or equal capacity from system α than B, or equivalently that B is at least as difficult to satisfy as A in α. A more constrained relationship between Tasks on α, named *covering*, is given as follows:

Definition 2 $\forall A, B \in Tasks,\ A \leq B, \nexists C \in Tasks, C \neq A, C \neq B, A \leq C \leq B \Leftrightarrow A \preceq B$.

It means α cannot provide such a capacity or resource which is in the medium of the need of A and B. For example, consider there is a multi-FPGA based cluster. Each FPGA can only be assigned to run one task at once and then it is easy to say a task which asks for two FPGAs simultaneously will actually covers a task which asks for only one FPGA as there is no any other tasks between them such as asking for one and a half FPGA.

When two tasks are uncorrelated - a parallel relationship between them is defined as following:

Definition 3 $\forall A, B \in tasks, A \not\leq B\ \wedge B \not\leq A\ \Leftrightarrow A \parallel B$.

Some kinds of tasks use only one type of hardware resources. For instance, one task, named A_1, may only need some CPU capacities for computation while another, named A_2, just need some memory to record information. It

is easy to verify that not all cluster devices which can serve A_1 will serve A_2 and vice versa. So in this case, A_1 and A_2 are parallel. Then it is easy to extend this instance to a more general case with n different dimensions of tasks, and there is a following corollary: $\forall i, j \in n, i \neq j \Leftrightarrow A_i \parallel A_j$

After that the union of tasks can be defined: some more complicated tasks may contain multi-dimensional resource needs, which can be viewed as asking different one-dimensional tasks simultaneously and this can be achieved by the union of those parallel A_i:

Definition 4 *Given A^n, a n-dimensional task, then: $A^n = (\forall A_i, i \in n) \bigvee A_i$.*

By defining relationships between tasks, a closed-topology for a general multi-FPGA system involving irregular tasks can be built. Any possible task can be viewed as either a one-dimensional task or the union of one-dimensional parallel tasks and the topology can be generated as a partially-ordered set (*poset*) through the relationships. In fact, it will be a *lattice* structure: there is a common lower bound of all tasks for *non-resource-need*, which is denoted by $\perp$. There is also a common upper bound of all tasks for any certain multi-FPGA system which asks for all the possible capacities simultaneously of this system.

A concrete example of this partial ordered model for multi-FPGA tasks targeting a 8-FPGA system is shown in figure 6.2. The 8 FPGAs in this example is connected by a ring topology where the first FPGA and the final FPGA are directly connected. In this figure, Ii denotes a task asking for i independent FPGAs and Ci denotes a task asking for i directly connected FPGAs.

6.2.2 Representing and Ranking Irregular tasks

Based on this model, suitable rankings on tasks to show the level of difficulty in allocating corresponding resource in general multi-FPGA systems is possible to be designed. An efficient way to construct rankings on modular lattice topologies is by considering the *height* function [86] which counts on the number of interim nodes from the bottom to the current node.

Definition 5 *$\forall$ x,y $\in$ lattice $\mathcal{L}$, a function H is called height iff:*

$$(1)\ H(x) = H(y) + 1 \ \Leftrightarrow y \preceq x;$$

$$(2)\ H(x) = 0 \ \Leftrightarrow x = \perp.$$

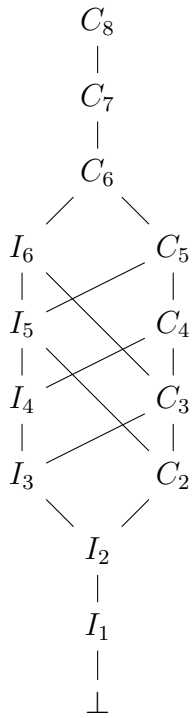


Figure 6.2: The partial ordered model of multi-FPGA tasks targeting a 8-FPGA system. The 8 FPGAs in this example is connected by a ring topology where the first FPGA and the final FPGA are directly connected. In this figure, Ii denotes a task asking for i independent FPGAs and Ci denotes a task asking for i directly connected FPGAs.

Universal consistency is satisfied if and only if the height function on lattice representation of tasks can label each node to result in a *consistent distance* between that node to the bottom whilst preserving their initial order. The consistence means for any node in the lattice, the length of every path from that node to the bottom will be the same. For a finite lattice, it is equivalent to state that the ranking R satisfy a *valuation law* [15]:

$$\forall x, y \in lattice\ \mathcal{L},\ R(x \cap y) + R(x \cup y) = R(x) + R(y).$$

The difficulty appears when considering the non-modular lattice structure for general multi-FPGA systems with arbitrary topology: if there is a node which has different interim nodes from itself to bottom by following different paths, then it cannot be easily ranked by the height function.

A concrete example of this scenario is provided as shown in the left hand side of figure 6.3, which is a subset of figure 6.2. Consider the node e ($I5$ in figure 6.2), its *height* value will either equal to 3 or 2 based on the different paths followed from the bottom node a ($I2$ in figure 6.2).

The lattice-based approach definition uses some definitions from set and lattice theory:

Definition 6 *Given a poset $\mathcal{P}$, a subset $\mathcal{S} \subseteq \mathcal{P}$ is a downset of $x \in \mathcal{P}$, named $\mathcal{S} = x{\downarrow}$ iff:*

$$\forall a \in \mathcal{P}\ , a\ \preceq x,\ a \in \mathcal{S}$$

The downset of a resource allocation request $x \downarrow$ is a set containing that request and all smaller requests which are less than x. i.e the downset of bottom element in a lattice contains only itself while the downset of top element in a lattice contains all elements in this lattice.

Definition 7 *Given a lattice $\mathcal{L}$, an element $x \in \mathcal{L}$ is join-irreducible iff:*

$$\forall a, b \in \mathcal{L}, x = a \sqcup b \leftrightarrow x = a\ or\ x = b$$

The original approach to solve this problem is by considering an reverse *Birkhoff's representation* [15] on the initial non-modular lattice topology for tasks model on general cluster:

Theorem 8 *(Birkhoff) Any finite modular[2] lattice L is isomorphic to the lattice of downsets of the partial order of the join-irreducible elements of L.*

[2]In Birkhoff's original definition, it used *distributed lattice* instead and gave a corollary saying any distributed lattice is modular. The followed result is used directly in this chapter as the detail mathematical description of the original theorems and corollary with the corresponding proof are far away beyond the scope of this thesis.

The verification of this theorem is presented in Birkhoff's well-known book [15]. This theorem efficiently provides a way to transfer any non-modular lattice, or widely say any poset, to a modular lattice composed by downsets of *join-irreducible* elements in the initial topology which can easily be ranked through the height function just defined. As for implementing the representation process, there are two steps: 1) generate the set of downsets for each task in the initial topology by traversing the initial set of orders between tasks. 2) generate the resulting modular lattice by ordering the downsets through inclusion relationships. Recall the example shown in Fig. 6.3, the resulting modular lattice is presented in the right hand side where each node in here has a *height* value. A theorem to guarantee *easy-computable* of the proposed process is that the height value for each downset in the resulting modular lattice is equal to the number of tasks it contains:

Theorem 9 $\forall X{\downarrow} \in$ *modular downset lattice* $\mathcal{L}'$ *containing* N *tasks in initial representation* $\mathcal{L}$, $H(X{\downarrow}) = N$.

Proof: Proof by induction: when N = 1, different X↓ contains single different task, so those X↓ are independent of each other and the bottom $\perp$ is covered by all of them. Then in this case, there is H(X↓) = 1.

Let H(X↓) = N for $N = k$. When $N = k + 1$, different X↓ contains one more task compared with when $N = k$. All of those different X↓ in this level cannot be included by each other as they have the same size, so they can only cover the downsets in a lower level. If a X↓ include one downset in the level below N=k, then there must be a corresponding downset in the N=k level which contains all the tasks of the lower level downset and be included in X↓. So all X↓ in N=$k+1$ level can only cover downset in N=k level, then there is: H(X↓)(N=k+1) = H(X↓)(N=k) + 1 which finish the verification.

□

Based on this theorem, the irregular tasks can be ranked by assigning the size of its corresponding downset without generating the modular lattice. A pseudo-code description of this process is shown in **Algorithm 6**. It shows that this ranking is *easy computable*, that the processing can be finished within polynomial time O(n^2) by only traversing the initial matrix of orders between tasks. The ranking value can be assigned for each task after generating the corresponding downset instead of traversing the set of tasks one more time after the result modular downset lattice generated.

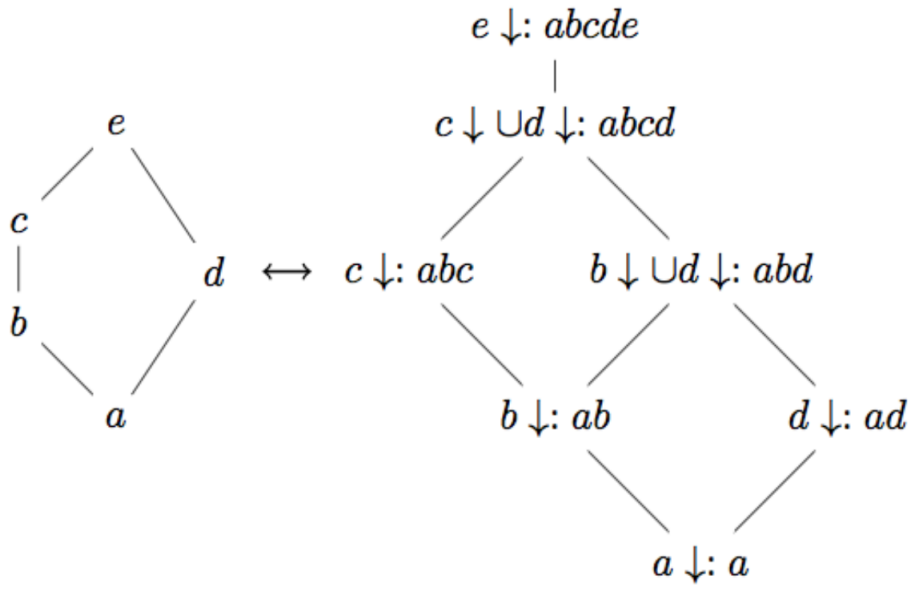


Figure 6.3: Lattice representation. The left hand side is a subset of the partial ordered model from figure 6.2, where a represents $I2$ asking for two independent FPGAs, b represents $I3$ asking for three independent FPGAs, c represents $I4$ asking for four independent FPGAs, d represents $C2$ asking for two directly connected FPGAs and e represents $I5$ asking for five independent FPGAs. Recall that in this example, the system composed by 8 FPGAs connected in a ring topology. So if a task asking for 5 independent FPGAs, there must be at least 2 FPGAs are directly connected, which leads to d(C2) $\preceq$ e(I5). The right hand side is the resulting modular downset lattice model. $a \sqcup b$ means a task asking for c and d simultaneously.

Algorithm 6 Birkhoff's representation to rank tasks

1: **INPUT:** (Set of tasks with dependences (taskOrders) // the initial partial ordered task model)
2: // Initialisation: setup the empty downset for each task
3: $N =$ *number of tasks*
4: **for** i = 1 to N **do**
5: Set $taskDownset_i = \{\}$
6:
7: // Representation: Birkhoff's representation by computing the size of downset of each task
8: **for** i = 1 to N **do**
9: **for** j = 1 to N **do**
10: **if** $taskOrders_j \leq taskOrders_i$ **then**
11: $taskDownset_i = taskDownset_i + 1$
12:
13: // Ranking: set the ranking value of each task to be its downset size
14: $taskRank_i = getSize.taskDownset_i$
15: **OUTPUT:** (downset based task model with rankings)

Finally, verify the correctness of ranking. The height function on downset lattice is *universally consistent*. Below shows that the ranking on height function is *order-preserving* for the initial relationship between tasks:

Theorem 10 $\forall$ *$X\downarrow, Y\downarrow \in$ modular downset lattice $\mathcal{L}'$ while X,Y are tasks in initial topology $\mathcal{L}$, $X \leq Y \Rightarrow H(X\downarrow) \leq H(Y\downarrow)$.*

Proof: $\forall$ X,Y $\in \mathcal{L}$, if X $\leq$ Y, then there is X$\downarrow \subseteq$ Y$\downarrow$, which leads to (X$\downarrow$) $\leq$ (Y$\downarrow$) in $\mathcal{L}'$. Recall the definition of height function, it is easy to say H is monotonous increasing and this give (X$\downarrow$) $\leq$ (Y$\downarrow$) $\Rightarrow$ H(X$\downarrow$) $\leq$ H(Y$\downarrow$). Based on the transitivity of the partial-order relation, there is: X $\leq$ Y $\Rightarrow$ H(X$\downarrow$) $\leq$ H(Y$\downarrow$).

□

In conclusion, a novel ranking method has been designed for dynamically scheduling tasks on FPGA-based multi-accelerator system. Unlike previous work, it achieves *universal consistency* for irregular multi-FPGA tasks. It is *easy computable* within polynomial time.

6.3 AccCOLAB Scheduling Algorithm

This section illustrates a dynamic scheduling algorithm equipped with the above ranking model.

Firstly, a problem formulation is provided, then an illustration of the terminologies used to denote variables in the algorithm and their initialisation. The main body of scheduling algorithm is illustrated by steps: fit and round. Finally, the algorithm is analysed by its complexity and theoretical performance upper bound.

6.3.1 Problem Formulation

Consider the underlining problem: there is a certain amount of hardware accelerator resources which can be formulated as a set of $\mathbb{A} : Acc_1...Acc_M$; a multi-workload scenario is denoted as $S : W_1...W_N$, where each workload W_k contains different number of tasks $\mathbb{T}_k : task_1...task_{N'}$. The outputs of the algorithm are scheduling solutions derived at runtime which can be formulated as mappings $\mathcal{P}(\mathcal{S}) : \{\mathbb{T}_k\} \mapsto \mathbb{A}$. Each workload has a completion time $wTime_k$:

$$wTime_k(\mathcal{P}) = max.\ taskTime_i(\mathcal{P}), i \in (1 \dots N')$$

Where $taskTime_i$ is the completion time of $task_i$. Based on the different mappings $\mathcal{P}$, the multi-FPGA system will have different total workloads completion time WCT which can be formulated as a higher level function of the mapping $\mathcal{P}$:

$$WCT(\mathcal{P}) = max.\ wTime_k(\mathcal{P}), k \in (1 \dots N)$$

The purpose of the algorithm is to give good solutions $\mathcal{P}$ to achieve high system performance represented by reducing WCT. The optimal solution $\mathcal{P}'$ on the given set $\mathbb{A}$ and $\mathbb{W}$ can be formulated as follows:

$$\forall \mathcal{P}(\mathcal{S}) : \{\mathbb{T}_k\} \mapsto \mathbb{A}, WCT(\mathcal{P}') \leq WCT(\mathcal{P}).$$

6.3.2 Abbreviations and Initialisation

This subsection presents abbreviations used in the algorithm description to illustrate the algorithm. As presented above, N is used to denote the number

Table 6.1: Abbreviations in Scheduling Algorithm and Initialization of Parameters

Abbreviations	
Acc	accelerator
AccState	whether accelerator is busy or not
AccTime	working time of accelerator
taskDim	number of accelerators asked by task
taskTopl	topology of accelerators asked by task
taskTime	completion time of task
taskState	whether task has been allocated or not
taskRound	round number of task
taskSol	the solution/mapping of scheduling

Initialisation	
Acc$_j$: j=1…M	
AccState$_j$	0
AccTime$_j$	0
task$_i$: i=1…N'	
taskDim$_i$	(input)
taskTopl$_i$	(input)
taskTime$_i$	(input)
taskState$_i$	0
taskRound$_i$	1
taskSol$_{ij}$	0

of workloads in a multi-workload scenario, N' to denote the number of tasks in a workload, Acc to denote accelerator and M to denote the number of accelerators. Each accelerator has a flag state (AccState) showing whether it is busy (1) or not (0). It also has a time state (AccTime) showing how long it should be working on the current task. Each input workload is presented by a set of tasks: Each task can ask for certain amount of accelerators (taskDim) and specific topology of them (taskTopl). A predicted task completion time (taskTime) for each task is also associated in the input workloads which is determined by an analysis. For each task, an allocation state (taskState) is used to represent whether it has been allocated to run on suited resources (1) or not (0) as well as a round number (taskRound). This round number shows the predicted starting time of the given tasks . A N'*M matrix (taskSol) is applied to record the solutions of scheduling for each workload, where $taskSol_{i,j} = 1$ means $task_i$ is assigned to run on accelerator Acc_j. Those terms are summarized in the left hand side of Table 6.1.

Before the running of the scheduler, all accelerators should be idle. Initially all tasks have not been scheduled to any accelerator. This initialization is shown in the right hand side of Table 6.1.

6.3.3 Ranking-based Scheduling

This section illustrates the main body the ranking-based heuristic scheduling algorithm. The pseudo code of is shown in Algorithm 7.

Fit Mechanism

The first step to implement the scheduling process is how to achieve the *fit* mechanism based on rankings from tasks. The proposed fit mechanism is a variant edition of the well-known First-Fit-Decreasing (FFD) approach guided by ranking. As demonstrated by [78], FFD is efficient for heuristics-based allocation on multi-resource environments. Recall that the ranking of a task is *universally consistent* and *easily computable*, which means it can identify the most difficult-to-satisfy irregular tasks in a given system efficiently in polynomial time.

Similar to a previous static multi-FPGA resource allocator [109], the ranking value of each task used as a heuristic to guide the allocator to make deterministic decision on selecting the most *difficult-to-satisfy* task, represented by the highest ranking value. Different from the static resource allocator, which only assigns resources at once and output static mapping, the runtime scheduler will assign newly released resources during runtime to those tasks which could not be allocated in the previous rounds, and in each round, it selects the most *difficult-to-satisfy* task with the highest ranking value first.

Round Mechanism

A Round mechanism is involved to finish the runtime scheduling. The scheduler traverses all remain tasks (taskState = 0) after each round until all tasks have been allocated to run on accelerators. In each round, if the task's round number (taskRound) is greater than some accelerator's expected execution time (AccTime), those accelerator will be released. If the task still cannot be allocated after new resources have been released, its round number will be added which means it delays to at least in next round. If the task can be allocated in the current round by scheduling on some accelerators, then for those accelerators, their expecting execution time should be increased by adding the predicted time of this task.

For example, there is a task which needs 2 interconnected accelerators for 2 time units and there are only 2 interconnected accelerators who are currently busy for other tasks and their expected working time is until the

Algorithm 7 Ranking-based Scheduling Algorithm on multi-FPGA Systems

INPUT: Set of task ordered by ranking
Initialization (refer to Tab.6.1)
// If there is any remaining task in queue
while N > 0 **do**
  // Schedule tasks based on ranking order
  for each $task_i$ in queue **do**
    if $taskState_i$=0 **then**
      //Release new resources if the current round number is greater than the accelerator's execution time.
      for j' $\in$ 0 to M **do**
        if $taskRound_i \geq AccTime_{j'}$ **then**
          $AccState_d$=0
      // Schedule $task_i$ to accelerators.
      // Count the number of accelerators allocated on the given task
      count = 0
      while count $\leq taskDim_i$ **do**
        if $AccState_j = 0$ **then**
          $taskSol_{i,j} = 1$
          $AccState_j = 1$
          $AccTime_j = AccTime_j + taskTime_i$
          count = count + 1
        j = j+1
      //If the current task can be scheduled, dequeue this task
      if j $\leq$ M **then**
        $taskState_i = 1$
        dequeue($task_i$)
        N = N -1
      else
        //If the current task cannot be scheduled, mask to next round
        $taskRound_i = taskRound_i$ +1
        reset $task_i$ states and release the accelerators on this task
      j $\rightarrow$ the first empty Acc in the first empty device
OUTPUT: Schedule of tasks presented in taskSol

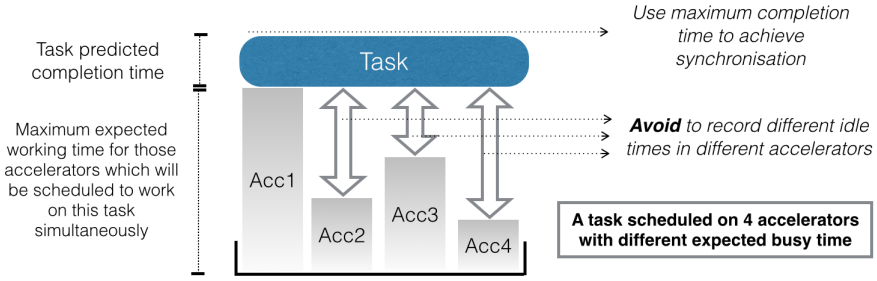


Figure 6.4: Abstract Model of multi-accelerator Synchronisation

3rd time units from now on. This new task should be waiting during the first 2 rounds and then in the 3rd round, the two accelerators will be released and it will be scheduled on those accelerators. The expected working time for those two accelerators will then be increased by adding 2 time units.

Another issue for common heuristic-based runtime scheduling is to avoid *task starvation*. A consequence of task starvation is idle resources and huge runtime overhead - if the scheduler gives a particular task a high priority to be scheduled, then low-ranking tasks maybe wait in the queue and cannot get a chance to be scheduled on available resources. The proposed round mechanism directly solves this problem based on the fact that it traverses all remaining tasks in each round before handling new tasks. If a task can be allocated to run in a certain time stamp during the runtime process, the scheduler is required to let this task run no later than the end of the corresponding round.

A technical problem in this approach is that during the runtime scheduling process, it is not easy to know how long each accelerator will be idle before it can be assigned to a new task. A technique applied to address this problem is that instead of trying to record the idle time of each accelerator in a given system through multiple counters during runtime, a global counter is applied to record the maximum expected execution time for each accelerator when they schedule on the same task and then apply a normalization process to achieve synchronization. An abstract model is shown in Figure 6.4 to illustrate this technique. A time unit can be set up to represent different real time slot based on the different predicted execution time of workloads. For instance, if a workload on a certain trace needs to work on several minutes on a targeting accelerator, then one time unit can be set up to one minute in the scheduler configuration.

6.3.4 Analysis

Complexity

The complexity for optimal task scheduling is known to be NP-hard [94]. Efficient heuristic based approach usually provides a polynomial time heuristic method to find reasonably good solutions.

Through each round of the runtime scheduling, the scheduler will only traverse the remaining tasks once, which leads to a complexity O(N) where N is the number of tasks. In practice, the algorithm can make the decisions of how to schedule hundreds of tasks within milliseconds which leads to an negligible scheduling overhead.

Performance Upper-Bound

Another issue is the theoretical upper bound of the performance in a perfect solution. Note the difference between the specific goal in here and a pure bin-packing problem: the bin-packing problem aims to reduce the number of bins used for certain number of balls while in this case, while in this case there are certain amount of bins and scheduler needs to reduce the loads for each bin which represents the performance. Based on this, the theoretical *d-approximation* upper bound for any heuristic-based algorithm on d-dimensional bin-packing problems [107] is not suitable for this work here, but a simple theoretical upper-bound targeting on performance can be defined as:

$$WCT_{upper} \geq max.\{\frac{\sum wTime_k}{M}, max.\{wTime_k\}\}$$

Where WCT_{upper} denotes the upper-bound of the system performance. In an optimal solution, WCT_{upper} cannot be less than either the total workloads completion time over the number of accelerators applied or the maximum of each individual workload completion time. This upper bound is applied to evaluate the proposed algorithm against FIFO and other heuristic-based approaches in the experiment section.

6.4 Software Architecture

The software architecture of this approach is shown in Fig. 6.5. The Runtime Scheduling system is responsible for the order of execution and the mapping

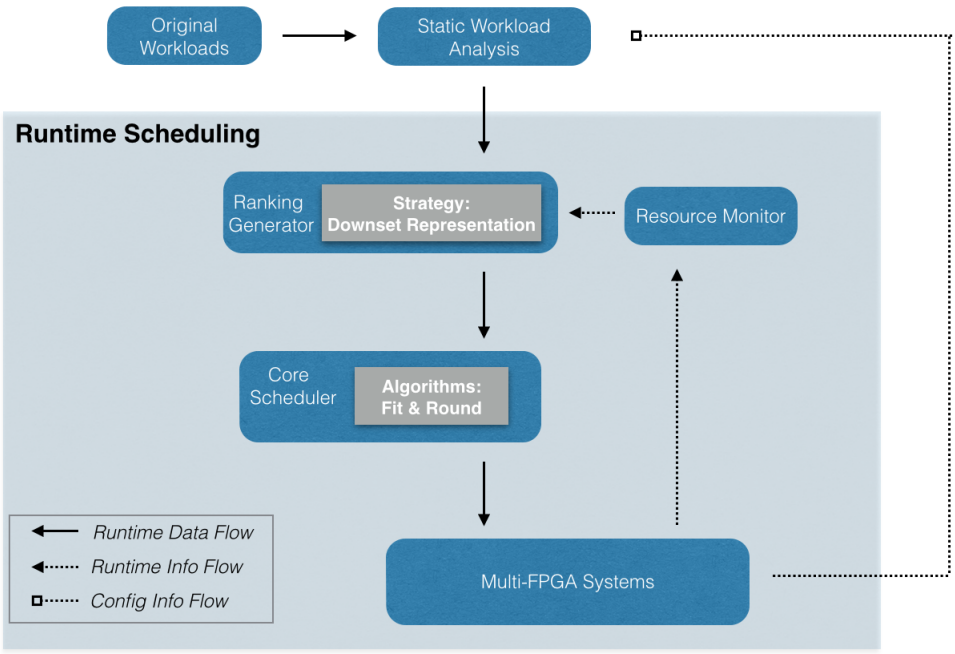


Figure 6.5: Scheduling Architecture

of tasks to FPGAs. The tasks are all *malleable*, meaning they can be instantiated on any number of compute units in parallel, and any number of tasks may be scheduled concurrently.

The original workloads from application traces initially flow into the static workload analysis. This offline process empirically determines and stores the ideal system configuration for each task in terms of compute unit allocation, targeting a specific hardware architecture employed, and assuming full resource availability. This approach is similar to that taken in [58, 99].

At runtime, the tasks are first ranked by the Ranking Generator. This decision is based on both the static information about the ideal allocation provided from the analysis, and runtime information about the current resource availability, provided by the Resource Monitor. Finally, the scheduler allocates tasks to FPGAs based on the Fit and Round algorithm.

6.4.1 Components

Main functional components of the software architecture are illustrated in this section. They are the Resource Monitor, the Ranking Generator and the Core Scheduler. Trivial interferences between those components within the main architecture are ignored.

- Resource Monitor: A basic component in this architecture is the resource monitor. The resource monitor records the runtime resource usage of the system and reports to the ranking generator. The runtime resource usage also containing the topology of those idle resources - whether interconnected or not. At runtime, if any hardware accelerator changes state, it will send a message to the monitor.

- Ranking Generator: The ranking generator receives the information from both workload analysis and resource monitor, and then computes the *universally-consistent* and *polynomial time computable* ranking values by applying the *downset* representation strategy and the ranking generation algorithm as described in **Algorithm 6**. This computation is triggered during runtime whenever new tasks are arrived.

- Core Scheduler: The core task scheduler is a novel component in this architecture. Instead of taking a FIFO approach, it applies a heuristic-based approach guided by ranking values to select tasks to schedule on

available resources. The underlining algorithm in this scheduler is implemented through the *Fit* and *Round* mechanism as described above in **Algorithm 7**. The output of this scheduler is the final scheduling decision which indicates the tasks to run on corresponding resources and can be viewed as a mapping from tasks to resources.

6.4.2 Analysis

This subsection analyses two main characteristics of the proposed scheduling approach.

Generality of Application

The generality of the proposed scheduling approach can be considered as follows:

- Varying workloads: The proposed analysis stage automatically adjusts to target different workloads, and different granularity of data input.
- Varying optimizations: The proposed analysis stage can adjust to target different optimization goals such as high performance, high throughput or high energy efficiency by setup specific metrics without further changing the underlining software architecture.
- Varying architectural setup: The scheduling approach optimises for different configurations of FPGA clusters, with different topologies and number of accelerators. Updating the domain of the proposed lattice representation for ranking generation does not lead to further change of the underlining representation algorithm.

Availability

Based on the workload analysis which can be equipped with different performance metrics, the proposed software architecture can be customized to achieve high performance for a wide class of HPC applications. The partial-order model and ranking-based heuristic are limited by the following assumptions:

- The links for adjust accelerators are stable. This approach does not consider reliability or fault-tolerant issues of links.

- Any multi-FPGA system with only four or fewer FPGAs is considered to be trivial for multi-FPGA scheduling. Simple scheduling algorithms should be sufficient in these cases – this work targets more complex networks of accelerators where all accelerators cannot be fully directly linked in a crossbar structure.

- This work only considers how to schedule workloads between accelerators across a multi-FPGA system. It does not consider the internal configuration of each FPGA for computation and single FPGA kernel optimisation – those are beyond the scope of this work.

In conclusion, the proposed approach is suited for wide classes of HPC applications/kernels to achieve application-specific goals targeting on current multi-FPGA systems.

6.5 Experimental Setup

This work was tested on a testbed with FPGA-based multi-accelerator systems, the MPC-X device produced by Maxeler Technologies [73]. It is evaluated by both the F11 Genetic algorithm benchmarks and the Reverse Time Migration (RTM) application. The hardware and workloads were described in the technical background chapter above. The proposed software resource manager and scheduler are implemented as a middleware on top of the system OS using Java.

The approaches of preferring real-world techniques over synthetic benchmarking make little sense in the context of evaluating irregular workloads, where the heterogeneous hardware configuration is chosen over homogeneous parallel machines precisely to adapt to the real problem of irregular workloads.

6.5.1 Static Analysis

A static analysis is completed to:

- Classify tasks and determine the best system configuration targeting the given hardware resources for each task.

- Classify the workloads and analyse the workload irregularity.

Tasks Classification

When the physical cluster is mainly composed of multi-FPGA systems as presented in Figure 6.5, the basic analysis metric is $taskTime_i(S_k)$, *the predicted execution time* for each task i on the system configuration S_k where k denotes the number of interconnected accelerators. The static analysis will then select the best system configuration S_k for each task i. Trivially, it selects a S_k to achieve the minimum of $taskTime_i(S_k)$ which represents the highest performance the system can reach.

Counterexamples also exist, such as if there are only a few hard tasks, why not just schedule them by each task per accelerator simultaneously to finish them as soon as possible and achieve an overall high performance? This counterexample would be more serious when there are workloads containing different sizes of tasks. If the scheduler can assign large tasks on multiple accelerators while basic small tasks run on each per accelerator, it may save a lot of resources, which can then be used to finish many other co-executed tasks. So instead of just applying the basic metric $taskTime_i(S_k)$, a customised metric and two special cases below are to be considered:

Normal Cases: A metric, named $dT_{task/Acc}(S_k, i)$ is to show the ratio of *performance gain* of task i when adding accelerators to reach the configuration S_k, which can be computed as follow:

$$dT_{task/Acc}(S_k, i) = \frac{taskTime(S_{k-1}, i) - taskTime(S_k, i)}{|S_k| - |S_{k-1}|}$$

Where $|S_k|$ is the number of accelerators in configuration S_k. Note that $|S_k|$ may not equal to k as there may be some non-connected accelerators in this configuration. The workload analysis selects the S_k which the maximum $dT_{task/Acc}(S_k, i)$ value which means the tasks enjoy the most significant performance gain after adding more accelerators to run on it to reach this configuration and if continue add accelerators on this task, the performance will not increase much. Although presenting execution time on variable number of FPGAs is a common approach to show performance of multi-FPGA systems by research communities [32, 24], researchers only use those kind of graphs to prove their scalability instead of extracting good configurations. Thus, this work extracts a new metric based on the initial motivation from typical mathematical

optimisation: the derivate of $T_{task/Acc}(S_k, i)$.[3]

Special Cases: A special *upper* case appears where a workload only contains a few large/hard tasks. For example, if a user submits a workload containing only one large task, the static analysis should allocate all possible resources to work on it simultaneously to reach best performance. The exact entry condition of the upper case will be different based on different application workloads. A special *bottom* case appears where a workload just contains many small/easy tasks. For example, if a user submits a workload just containing several small tasks, the workload analysis should allocate available resources to work on them simultaneously (one accelerator per task) to reach the overall best performance. The exact conditions for a trace to be classified in this case will be different based on different application workloads.

As a result of those cases, the final performance metric $m(S_k, i)$ used to select best configuration in the workload analysis is selected as a trade-off:

$$m(S_k, i) = \begin{cases} all\ Accs\ per\ task & \text{for} \quad upper\ case \\ one\ Acc\ per\ task & \text{for} \quad bottom\ case \\ dT_{task/Acc}(S_k, i) & \text{for} \quad o.w. \end{cases}$$

The output information from this static analysis will include the tasks with corresponding configuration information:

$$S_k = \{taskDim, taskTopl, taskTime\}$$

As shown above, it includes the required number of accelerators $taskDim$, the topology of those accelerators $taskTopl$ and the corresponding predicted execution time $taskTime$ based on this configuration.

In the experiments, there are three different sizes of workloads for both RTM tasks and GA F11 benchmark input: small (S), middle (M) and large (L). Multi-large tasks scenarios (Multi-L) for RTM and Mix-large domain scenarios (Mix-L) for GA F11 benchmark are added to increase the complexity and irregularity of the workloads. Each case is run by four times and then the mean value of execution time which is viewed as the original $taskTime_i(S_k)$.

[3]Different with the real derivate, it does not need to consider a δT in the discrete case here.

Table 6.2: Static Analysis

Workload Preprocessing of RTM Jobs

Size	Best Configuration	Execution Time
S	1–G	40s
M	2–R	5m 15s
L	4–R	1h 10m 56s
Multi-L	2–R	41m 17s

Workload Preprocessing of F11 GA Benchmark

Size	Best Configuration	Execution Time
S	2–R	11s 200ms
M	3–R	30s 500ms
L	4–R	53s 300ms
Mix-L	2–R	1m 3s 900ms

Different system configurations use different number of FPGAs and topologies between those FPGAs including Ring and Group. A Ring type denotes connected multi-FPGA whilst Group type denotes independent FPGAs in one device. The analysis only shows one FPGA case for the Group type topology based on the following trivial equation:

$$taskTime_i(1\_G) = taskTime_i(k\_G), k \in [1...5]$$

Where k_G means the configuration S_k fully composed by k FPGAs in a Group type. This equation trivially shows that if running one task on 1 FPGA, then its running time should be equal to run k task on k Group FPGAs simultaneously. Recall the hardware limitations, k should not be greater than 5 in the experiment. Similar as for Group type, k_R is used to denote the configuration S_k fully composed by k FPGAs in a Ring type.

The best configuration of each workload size for both applications with corresponding average execution time are presented in Table 6.2.

Workloads Classification

The static analysis performed allows workload-level classification. Recall the motivating example in introduction section – the irregularity of tasks presented the difference between suitable configurations, the number of accelerators and the topology needed. It is hard to define the underlining irregularity from attributes of different data directly as it also depends on the workloads composition and kernels which are actually used to execute it.

Instead, an empirical metric of irregularity in workloads is given here to demonstrate the results. It represents how irregularity between different given tasks caused slowdown in the experiments. In the most irregular case, each different class (*small,medium or large*) of tasks has the same amount of total predicted execution time which leads to the same importance on overall

performance. While in a case with low heterogeneity level, a certain class of tasks need much more total execution time than other tasks, and then the overall performance will more dependent on this class of tasks. A weighted edition of *the variance of execution time from each task* $Irg(W)$ to is applied to denote the irregularity level of each workload W_k as follows:

$$Irg(W_k) = \sum_{i=1}^{N'} (\frac{taskTime_i}{\sum taskTime_i}) * (\frac{taskTime_i - \overline{taskTime}}{\overline{taskTime}})^2$$

Where $\sum taskTime_i$ denotes the total execution time. The $\overline{taskTime}$ denotes the average execution time. N' denotes the total number of tasks in this workload.

After the analysis, the main runtime scheduler can begin to work and its performance is compared with a FIFO approach and the re-implemented HEFT and MFIT schedulers on both F11 and RTM workloads. The proposed approach is evaluated based on three kinds of multi-workload scenarios:

Single-task workloads

Each workload is made up of a single run of the benchmark, which consists of a number of parallel elements of varying irregularity.

Multi-task workloads

Each workload is made up of a random number of runs of the benchmark between 2 and 20, operating on different data. Each run consists a number of parallel elements of varying irregularity.

High irregularity workloads

The scheduler is evaluated with a multi-task scenario where the tasks considered are highly irregular in size.

6.6 Experimental Results

The main experimental results are presented in this section. The proposed ranking-based approach is compared against three other approaches – FIFO, HEFT, MFIT in three multi-workload scenarios – single-task, multi-task and

high irregularity. It compares principally against FIFO as it is the policy used in the Maxeler Orchestrator [73] commercial implementation.

Additionally, it is compared against an implementation of the the multi-FPGA research scheduling algorithm MFIT [59], and an implementation of the heterogeneous scheduling algorithm, HEFT [97], adapted for multi-FPGA scheduling. This is to show how this work compares to more sophisticated research-level approaches.

The results are presented as a percentage of the theoretical upper bound of performance, derived by assuming linear scaling of performance with number of FPGAs, described in the section above.

6.6.1 Results for Multi-task Cases

The experimental results of RTM workloads and GA F11 benchmark on different workload scenarios through different irregularity levels.

Each point in the plotted resulting graphs is the mean value of 50 tests to improve the quality of results. For example, the result for the *Ranking* approach in multi-task scenarios at the point of 0.4 normalized irregularity is the mean value of 50 points for which normalized irregularity ranges from 0.375 to 0.425. The reason the number 50 is selected for each point is just an empirical cost-efficiency value: Based on the initial tests, the mean values for around 10 to 30 times tests are fluctuated whilst the values from 50 times to more than 100 times are similar.

The *Ranking* approach consistently outperforms all other approaches in multi-task scenarios across levels of irregularity. For RTM workloads, *Ranking* maintains 99.7% to 99.8% of upper-bound of performance across levels of irregularity. This represents a significant improvement of between 5% and 7% over the commercially implemented FIFO scheme.

The re-implementation of the HEFT approach for this task performs fairly well here, but only reaches 97.5% to 98% of the maximum possible performance. MFIT performs similarly to FIFO.

For F11 workloads, with normalised irregularity around 0.3, *Ranking* already reaches more than 95% of the upper bound, while both FIFO and MFIT are around 92% and HEFT 87%. As irregularity increases to 0.55, the *Ranking* approach obtains 98% and does not decrease. Both FIFO and MFIT reach a maximum of 93.5%, while in this case, HEFT only obtains between 86% and 88% of the upper bound.

Over both scenarios, the *Ranking* approach provides the best performance.

6.6.2 Results for Single-task Cases

Although this work focuses on multi-task workloads, it also works well on single-task workloads. In single-task scenarios, the *Ranking* approach performs identically to HEFT, while outperforming both FIFO and MFIT. In practice, *Ranking* and HEFT make identical scheduling decisions in these simpler scenarios.

Ranking and HEFT obtain a 93.5% for the RTM workload when normalized irregularity is 0.35, while still outperforming both FIFO and MFIT, which obtain 89.8% and 89.4% respectively. Similar results are obtained with single-task F11 workloads.

6.6.3 Results for varying workload numbers in High Irregularity Scenarios

Figure 6.7 shows the results where only highly irregular workloads are considered, and where the number of workloads simultaneously present for computation is varied.

In general, scheduling becomes easier as the number of workloads increases as there are more tasks of varying size which allow holes in the schedule to be more easily filled. This intuition is borne out across all considered approaches.

Ranking generally performs better than all other approaches across both RTM and F11 workloads.

For RTM workloads, at a low number of workloads, HEFT performs a little better than ranking. For example, when the number of workload is around 30, HEFT can reach around 93% whilst *Ranking* only get around 85%. This is due to the HEFT algorithm preferring longer running tasks running on a small number of accelerators, rather than ranking, which prefers shorter running tasks running on a larger number of accelerators. The overhead of changing tasks more quickly in low workload size scenarios leads to lower performance with the ranking approach. Even in this case, *Ranking* is still outperforms MFIT and FIFO, both of which obtain about 76% of the available performance. *Ranking* consistently outperforms HEFT as the number of workloads increases past 50,quickly achieves 99% or available performance.

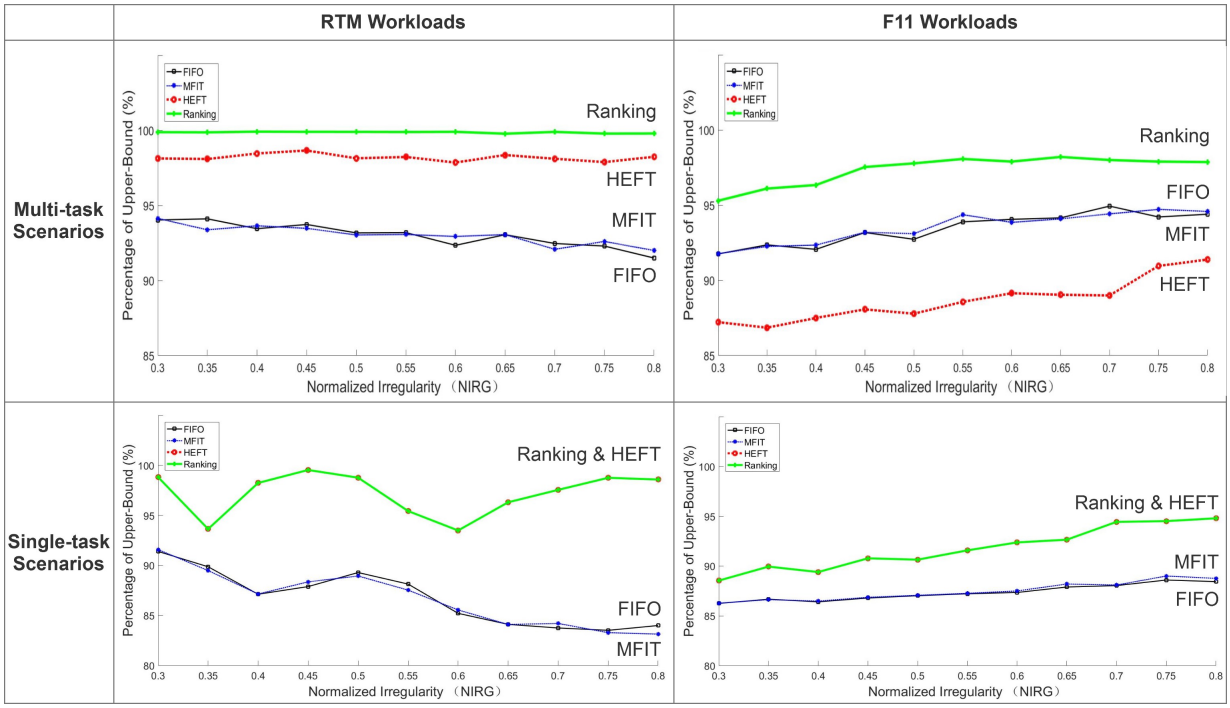


Figure 6.6: Experimental results on different workload scenarios

For F11 workloads, *Ranking* outperforms all other three approaches across all numbers of workload. The results vary from 91% to more than 98% of the upper bound of performance. MFIT and FIFO achieve around 93% when the number of workloads is greater than 60. HEFT fluctuates around 90% across differing numbers of workload.

6.7 Conclusion

In conclusion, this chapter presents a novel dynamic scheduling approach for multi-FPGA based clusters, based on a partial order representation of tasks and a ranking methodology.

It shows how the proposed dynamic scheduling approach, which models and ranks irregular workloads, outperforms existing approaches including FIFO and two heuristic-based research schedulers, HEFT and MFIT. The performance gains between the proposed approach against other approaches are more significant for high irregularity and multi-task scenarios on both

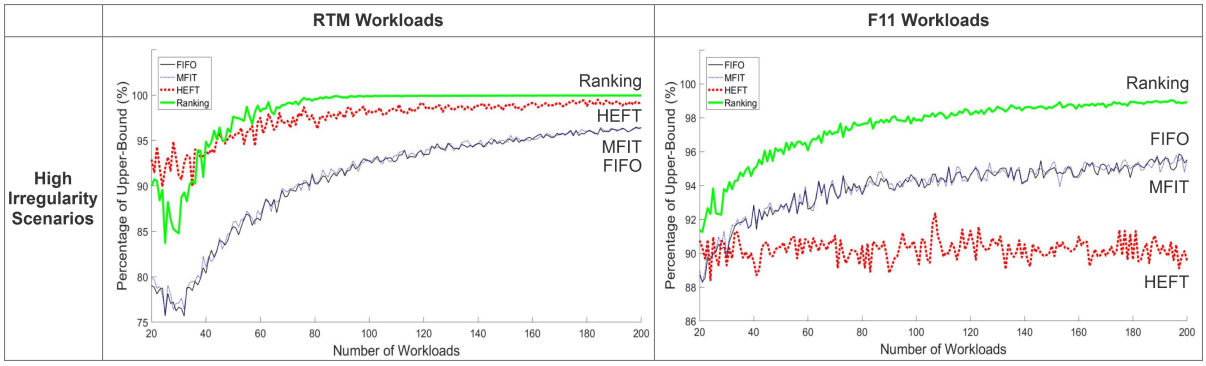


Figure 6.7: Experimental results on high irregularity scenarios

workloads. In scenarios where most workloads contain similar sizes of task, the proposed approach maintains the same or better performance. In these simpler cases, the existing approaches are already perform well.

This work also shows the prevalence of FIFO schedulers in industrial and research accelerator systems is not unreasonable – FIFO performs well in a mostly homogeneous single-task environment. However, in the presence of irregular tasks and FPGA-based multi-accelerator resources, a better method is required to retain high levels of performance.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Addressing heterogeneity is a critical issue when running complex workloads on modern computing systems, in order to guarantee good performance. Schedulers in operating systems have the main responsibility to assign software threads and allocate hardware resources in and efficient way. There is a clear need for such schedulers to be intelligently designed in a heterogeneity-aware way. This design is difficult because different types of heterogeneous systems have different heterogeneities, and a simple uniform approach cannot provide optimal solutions. Similarly, typical workloads on different types of systems have varying characteristics too. Adding these complexities up leads to a hugely complex and difficult optimisation space. Research in this area should investigate both heterogeneities – hardware resources and the target workloads – to design efficient schedulers.

This thesis presents a set of systematic methods to design heterogeneity-aware schedulers to accelerate system performance targeting varying major modern parallel machines, from FPGA-based clusters and hierarchical many-core supercomputers to asymmetric multi-core processors. It provides concrete methods to model and address the different heterogeneities from the scheduler side, including the asymmetric physical topology on multi-FPGA clusters, the hierarchical processors and memory architectures on supercomputers and the different computing ability and core sensitivity with fairness on big.LITTLE multi-core processors.

7.1.1 Contributions

This thesis makes four main contributions as follows:

- It analyses the general underlying problem models of heterogeneity-aware scheduling with data partitioning. It then provides efficient problem representations to guide the scheduler and data partitioner design and implementation.

- It investigates and develops concrete schedulers, either in the OS kernel (Linux) or middleware (FPGA resource manager) levels, targeting heterogeneous systems with multicore processors and accelerators.

- It investigates and develops data partitioning methods, mainly by redesign of the parallel algorithms used in large-scale workloads, targeting high performance supercomputers with hierarchical hardware architectures.

- It provides empirical studies and evaluation on either full system simulators or real machines to demonstrate the claimed advantage of the proposed heterogeneity-aware approaches on both benchmark workloads and scientific applications.

These four contributions are further summarised in the subsections below.

7.1.2 Heterogeneity Analysis

This thesis describes and discusses the general underlying problem to model different heterogeneities during runtime scheduling and guide the data partitioner. These include the asymmetric physical topology on multi-FPGA clusters, the hierarchical processors and memory architectures on supercomputers and the different computing ability and core sensitivity with fairness on big.LITTLE multi-core processors.

This thesis analyses the problem of how heterogeneity affects the decisions and performance of schedulers, and goes on to investigate the opportunites to address it. It also describes the problem of designing efficient hierarchical data partitioning methods compared with a basic parallel approach with simple dataflow partitioning, before discussing the potential improvement space available, based on the hardware heterogeneity.

7.1.3 Scheduler Design and Implementation

The proposed schedulers provide design principles and implementations to address the heterogeneities from different types of heterogeneous systems.

The scheduler running asymmetric multi-core processors for mobile devices is the first the first ever collaboration-based approach to address all three main runtime factors: *core sensitivity, thread criticality* and *fairness*, simultaneously. Compared with previous research on heterogeneity-aware schedulers, it presents a flexible framework to:

1. Accelerate multiple critical threads from multiple programs in-place on heterogeneous processors;

2. Control the thread scheduling more accurate by updating the linux kernel directly to scale time-slice and update task priorities instead of only effect the thread-to-core affinity by middleware or VM.

The novelty of this approach can be summarised as *multi-bottleneck co-acceleration*. Instead of putting all the pressure of bottleneck acceleration on the out-of-order high frequency *big* cores, it shows the benefit of letting the in-order low frequency $LITTLE$ cores also execute suitable threads intelligently to achieve multi-core collaboration during execution time. This scheduler has been implemented on the fully system simulator, GEM5 with interfaces to Linux kernel.

The scheduler for multi-FPGA clusters proposes a novel framework to address the heterogeneity generated by connection topology and computing ability. Instead of the fully-connected or crossbar topology used on small on-chip multi-core for CPUs and streaming multiprocessors for GPUs, multiple FPGAs in a cluster are usually physically connected by either a line or a ring with an InfiniBand-like connection system. This results in a specialised heterogeneity for multi-FPGA systems – communication-intensive multi-threaded tasks required to be allocated onto multiple physically connected FPGAs for frequent communications, while other computing-intensive tasks are free to be allocated onto multiple randomly located FPGAs to run individually without heavy communication. This thesis suggests a lattice-based representation and ranking heuristic to address this topological heterogeneity, together with consideration for the difference in resource need from applications. The lattice based model is applied to construct a consistent ranking between tasks, requesting a different amount of computing

resources and changing hardware topologies depending on the task. The runtime scheduler algorithm is equipped with this ranking heuristic to schedule tasks efficiently with negligible system overhead – the output table from lattice representation can be read in O(1) time.

7.1.4 Data Partitioner Design and Implementation

Regarding the application-specific data partitioning and scheduling targeting hierarchical many-core supercomputers, this thesis proposes an automatic and large-scale approach to map the parallel program onto the heterogeneous memory hierarchy. It investigates two concrete mappings:

1 Map the asymmetries on the frequency of synchronisation between multiple co-executed threads onto the heterogeneities from the communication speed between processors in different hardware architectures.

2 Map the asymmetries on the loads between different co-executed threads onto the heterogeneities from storage limitations between local caches and shared memory.

The proposed implementation uses on-chip multi-core processors with the highest available bandwidth and communication speed to partition the multiple dimensions of each data sample. It uses multiple core groups and distributed nodes with less bandwidth to partition the original dataflow since the communication need between multiple data is much less than the need to communicate between the multiple dimensions inside each data point. After the customised data partitioning, the runtime scheduler on supercomputers is can be greedy designed to find the good scheduling solutions.

7.1.5 Impact on Benchmark Workloads and Scientific Applications

The proposed scheduler for asymmetric multicore processors is tested on complex multi-threaded multi-programmed workloads, which are composed by systematically selected programs from benchmark suites including PARSEC3.0 and SPLASH-2. The experimental results on GEM5 simulator with simulated ARM A57/A53 architectures demonstrate that this approach gives the greatest benefit on complicated mix-workload scenarios with limited and

realistic processors resources, e.g. for smartphones, where the system needs to run multi-threaded multi-programmed workloads on limited number of cores. The proposed scheduler can outperform other work by up-to 27% on synchronisation-intensive workloads and 15% on communication-intensive workloads in terms of system throughput and normalised turnaround time. Its potential impact has attracted industrial attentions to re-implement on commercial chips for mobile devices, such as the Huawei Kirin 960.

The proposed data partitioner on heterogeneous manycore supercomputers is tested on large-scale workloads from both well-known machine learning benchmarks (ImgNet and UCI) and real applications (land cover and gene expression classification). Demonstrated by the concrete case study of k-means clustering based real applications on the Sunway Taihulight supercomputer with SW26010 many-core processors, this design surpasses the previous limitations on the tractable data size for clustering and shows a significant scalability by efficiently applying more than 1 million cores simultaneously to execute massive workloads. After the customised data partitioning, the runtime scheduler is designed to greedily find good solutions for scheduling these asymmetric software threads onto the supercomputer.

The proposed scheduler on multi-FPGA cluster is tested on mixed workloads from a genetic algorithm benchmark (F11) and an oil and gas industrial application (Reverse Time Migration). The experimental results demonstrate the claimed advantage of the proposed approach on multi-FPGA based Maxeler Dataflow Engines (MPC-X) with Xilinx FPGAs. It achieves up-to 20% performance gain in term of total execution time compared with other multi-FPGA schedulers and the default first-in-first-out based commercial scheduler.

7.2 Future Work

Future work is proposed on multi-accelerator based heterogeneous systems beyond FPGAs. A typical kind of accelerators in modern heterogeneous system which has not been covered by this work are the general-purpose Graphics Processing Units (GPUs).

GPUs are increasingly widely used to accelerate general-purpose computation. GPUs achieve high computational power by exploiting thread-level parallelism across a number of streaming multiprocessors (SMs). The number of SMs increases with each successive generation. Whereas Nvidia Fermi

and Kepler architecture implement 14 SMs (Tesla M2050) and 15 SMs (Tesla K40), respectively, the next-generation Maxwell has 24 SMs (Tesla M40), and the latest Pascal and Volta architectures feature as many as 56 SMs (Tesla P100) and 80 SMs (Tesla V100), respectively.

This hardware trend coincides with the application trend towards using GPUs as accelerators in cloud services and data centres to meet the ever-growing demands while maintaining cost efficiency, as exemplified by Amazon EC2's offering of GPU instances in the cloud. In such systems, multiple independent applications from different users need to be executed as efficiently as possible. Spatial multitasking is a promising solution to support GPUs multitasking where the SMs in a GPU are divided into disjoint subsets to which different kernels are assigned to co-run. For kernels with different resource demands, i.e., a memory-bound kernel and a compute-bound kernel, co-executing them on GPUs better utilizes the compute resources and memory resources, which can brings 63% performance improvement on average [119].

Current GPUs multitasking work does not fit the real world where the kernels executing on GPUs are actually launched by the host side – the CPU side. In other words, state-of-art GPUs multitasking all naively assumes that there some kernels waiting for GPUs to execute. Unfortunately, what kernels that GPUs can execute actually depends on the CPU scheduling policy.

A proposed future work to address this problem is to design a CPU/GPU co-scheduling framework using a compiler-based predictive model, and collaboratively aware CPU and GPU schedulers. In particular, the compiler-based predictive model could consist of deep learning based code segments classification which could analyses the GPU kernel characteristics within the application. Based on the kernel information, the CPU scheduler chooses the right application to execute and launch the suitable kernels to the GPU side. Finally, the GPU scheduler can exploits the GPUs multitasking to co-execute different GPU kernels and achieve the best performance.

Bibliography

[1] John Aldrich and Jeff Miller. Earliest known uses of some of the words of mathematics. *particular, the entries for" bell-shaped and bell curve"," normal (distribution)"," Gaussian", and" Error, law of error, theory of errors, etc*, 2014.

[2] Mauricio Araya-Polo et al. Assessing accelerator-based hpc reverse time migration. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 22(1):147–162, 2011.

[3] ARM. http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0388e/ behedihi.html. In *ARM Cortex-A57 Technical Reference Manual*, 2016.

[4] Mihai Bādoiu, Sariel Har-Peled, and Piotr Indyk. Approximate clustering via core-sets. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 250–257. ACM, 2002.

[5] Sudarshan Banerjee, Elaheh Bozorgzadeh, and Nikil Dutt. Considering run-time reconfiguration overhead in task graph transformations for dynamically reconfigurable architectures. In *13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'05)*, pages 273–274. IEEE, 2005.

[6] Michela Becchi and Patrick Crowley. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd conference on Computing frontiers (CF)*. ACM, 2006.

[7] Tobias Becker, Oskar Mencer, Stephen Weston, and Georgi Gaydadjiev. Maxeler data-flow in computational finance. In *FPGA Based Accelerators for Financial Applications*, pages 243–266. Springer, 2015.

[8] Amir Ben-Dor, Ron Shamir, and Zohar Yakhini. Clustering gene expression patterns. *Journal of computational biology*, 6(3-4):281–297, 1999.

[9] Michael A Bender, Jonathan Berry, Simon D Hammond, Branden Moore, Benjamin Moseley, and Cynthia A Phillips. k-means clustering on two-level memory systems. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 197–205. ACM, 2015.

[10] Janki Bhimani, Miriam Leeser, and Ningfang Mi. Accelerating k-means clustering with parallel implementations and gpu computing. In *High Performance Extreme Computing Conference (HPEC), 2015 IEEE*, pages 1–6. IEEE, 2015.

[11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, January 2011.

[12] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[13] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. The parsec benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel Architectures and Compilation Techniques (PACT)*, 2008.

[14] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[15] Garrett Birkhoff. *Lattice theory*, volume 25. American Mathematical Soc., 1940.

[16] Christian Böhm, Martin Perdacher, and Claudia Plant. Multi-core k-means. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 273–281. SIAM, 2017.

[17] Carlos Boneti, Roberto Gioiosa, Francisco J Cazorla, Julita Corbalan, Jesus Labarta, and Mateo Valero. Balancing hpc applications through

smart allocation of resources in mt processors. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–12. IEEE, 2008.

[18] Carlos Boneti, Roberto Gioiosa, Francisco J Cazorla, and Mateo Valero. A dynamic scheduler for balancing hpc applications. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 41. IEEE Press, 2008.

[19] Thomas Bottesch, Thomas Bühler, and Markus Kächele. Speeding up k-means by approximating euclidean distances via block vectors. In *International Conference on Machine Learning*, pages 2578–2586, 2016.

[20] Y Dora Cai, Rabindra Robby Ratan, Cuihua Shen, and Jay Alameda. Grouping game players using parallelized k-means on supercomputers. In *Proceedings of the 2015 XSEDE Conference: Scientific Advancements Enabled by Enhanced Cyberinfrastructure*, page 10. ACM, 2015.

[21] Ting Cao, Stephen M Blackburn, Tiejun Gao, and Kathryn S McKinley. The yin and yang of power and performance for asymmetric hardware and managed software. In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[22] Moses Charikar, Chandra Chekuri, Tomás Feder, and Rajeev Motwani. Incremental clustering and dynamic information retrieval. *SIAM Journal on Computing*, 33(6):1417–1440, 2004.

[23] Siya Chen, Tieli Sun, Fengqin Yang, Hongguang Sun, and Yu Guan. An improved optimum-path forest clustering algorithm for remote sensing image segmentation. *Computers & Geosciences*, 112:38–46, 2018.

[24] Yuk-Ming Choi and Hayden Kwok-Hay So. Map-reduce processing of k-means algorithm with fpga-accelerated computer cluster. In *ASAP*, pages 9–16. IEEE, 2014.

[25] Kallia Chronaki, Alejandro Rico, Marc Casas, Miquel Moretó, Rosa M Badia, Eduard Ayguadé, Jesus Labarta, and Mateo Valero. Task scheduling techniques for asymmetric multi-core systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(7):2074–2087, 2017.

[26] CloudLightning. Cloudlightning position paper. 2016.

[27] Guy Barrett Coleman and Harry C Andrews. Image segmentation by clustering. *Proceedings of the IEEE*, 67(5):773–785, 1979.

[28] David A Coley. *An introduction to genetic algorithms for scientists and engineers*. World scientific, 1999.

[29] Jose GF Coutinho, Mark Stillwell, Katerina Argyraki, George Ioannidis, Anca Iordache, Christoph Kleineweber, Alexandros Koliousis, John McGlone, Guillaume Pierre, Carmelo Ragusa, et al. The harness platform: A hardware-and network-enhanced software system for cloud computing. In *Software Architecture for Big Data and the Cloud*, pages 323–351. Elsevier, 2017.

[30] Xiaoli Cui, Pingfei Zhu, Xin Yang, Keqiu Li, and Changqing Ji. Optimized big data k-means clustering using mapreduce. *The Journal of Supercomputing*, 70(3):1249–1259, 2014.

[31] Ryan R Curtin. A dual-tree algorithm for fast k-means clustering with large k. In *Proceedings of the 2017 SIAM International Conference on Data Mining*, pages 300–308. SIAM, 2017.

[32] Guohao Dai et al. Foregraph: Exploring large-scale graph processing on multi-fpga architecture. In *FPGA*, pages 217–226. ACM, 2017.

[33] Ilke Demir, Krzysztof Koperski, David Lindenbaum, Guan Pang, Jing Huang, Saikat Basu, Forest Hughes, Devis Tuia, and Ramesh Raskar. Deepglobe 2018: A challenge to parse the earth through satellite images. *ArXiv e-prints*, 2018.

[34] Inderjit S Dhillon and Dharmendra S Modha. A data-clustering algorithm on distributed memory multiprocessors. In *Large-scale parallel data mining*, pages 245–260. Springer, 2002.

[35] Yufei Ding, Yue Zhao, Xipeng Shen, Madanlal Musuvathi, and Todd Mytkowicz. Yinyang k-means: A drop-in replacement of the classic k-means with consistent speedup. In *International Conference on Machine Learning*, pages 579–587, 2015.

[36] Kristof Du Bois, Stijn Eyerman, Jennifer B Sartor, and Lieven Eeckhout. Criticality stacks: Identifying critical threads in parallel programs using synchronization behavior. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[37] Alejandro Duran, Marc Gonzalez, and Julita Corbalán. Automatic thread distribution for nested parallelism in openmp. In *Proceedings of the 19th annual international conference on Supercomputing*, pages 121–130. ACM, 2005.

[38] Fernando A Endo, Damien Couroussé, and Henri-Pierre Charles. Micro-architectural simulation of in-order and out-of-order arm microprocessors with gem5. In *Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), 2014 International Conference on*, pages 266–273. IEEE, 2014.

[39] Stijn Eyerman and Lieven Eeckhout. System-level performance metrics for multiprogram workloads. *IEEE micro*, 28(3), 2008.

[40] Yuping Fan, Zhiling Lan, Paul Rich, William E Allcock, Michael E Papka, Brian Austin, and David Paul. Scheduling beyond cpus for hpc. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 97–108. ACM, 2019.

[41] Expression Project for Oncology (expO). http://www.intgen.org/.

[42] Haohuan Fu et al. Scaling reverse time migration performance through reconfigurable dataflow engines. *IEEE Micro*, 34(1):30–40, 2014.

[43] Haohuan Fu, Junfeng Liao, Jinzhe Yang, Lanning Wang, Zhenya Song, Xiaomeng Huang, Chao Yang, Wei Xue, Fangfang Liu, Fangli Qiao, et al. The sunway taihulight supercomputer: system and applications. *Science China Information Sciences*, 59(7):072001, 2016.

[44] Andrey Goder, Alexey Spiridonov, and Yin Wang. Bistro: Scheduling data-parallel jobs against live production systems. In *2015 {USENIX} Annual Technical Conference*, pages 459–471, 2015.

[45] Sudipto Guha, Adam Meyerson, Nina Mishra, Rajeev Motwani, and Liadan O'Callaghan. Clustering data streams: Theory and practice. *IEEE transactions on knowledge and data engineering*, 15(3):515–528, 2003.

[46] Ali Hadian and Saeed Shahrivari. High performance parallel k-means clustering for disk-resident datasets on multi-core cpus. *The Journal of Supercomputing*, 69(2):845–863, 2014.

[47] Greg Hamerly. Making k-means even faster. In *Proceedings of the 2010 SIAM international conference on data mining*, pages 130–140. SIAM, 2010.

[48] Jian-Jun Han, Xin Tao, Dakai Zhu, Hakan Aydin, Zili Shao, and Laurence T Yang. Multicore mixed-criticality systems: Partitioned scheduling and utilization bound. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 37(1):21–34, 2018.

[49] Manish Handa and Ranga Vemuri. An integrated online scheduling and placement methodology. In *International Conference on Field Programmable Logic and Applications*, pages 444–453. Springer, 2004.

[50] HARNESS. The harness platform: A hardware- and network-enhanced software system for cloud computing. Technical report, November 2015.

[51] Juan Mario Haut, Mercedes Paoletti, Javier Plaza, and Antonio Plaza. Cloud implementation of the k-means algorithm for hyperspectral image analysis. *The Journal of Supercomputing*, 73(1):514–529, 2017.

[52] N Heath. Azure: How microsoft plans to boost cloud speeds with an fpga injection. *TechRepublic*, 2016.

[53] Chien-Chun Hung, Ganesh Ananthanarayanan, Leana Golubchik, Minlan Yu, and Mingyang Zhang. Wide-area analytics with multiple resources. In *Proceedings of the Thirteenth EuroSys Conference*, page 12. ACM, 2018.

[54] ImgNet ILSVRC2012. http://www.image-net.org/challenges/lsvrc/2012/.

[55] Anil K Jain and Richard C Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., 1988.

[56] Brian Jeff. big.little technology moves towards fully heterogeneous global task scheduling. In *ARM White Paper*, 2013.

[57] Daxin Jiang, Chun Tang, and Aidong Zhang. Cluster analysis for gene expression data: a survey. *IEEE Transactions on knowledge and data engineering*, 16(11):1370–1386, 2004.

[58] Ivan Jibaja, Ting Cao, Stephen M Blackburn, and Kathryn S McKinley. Portable performance on asymmetric multicore processors. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO)*, 2016.

[59] Chao Jing et al. Energy-efficient scheduling on multi-fpga reconfigurable systems. *Microprocessors and Microsystems*, 37(6):590–600, 2013.

[60] José A Joao, M Aater Suleman, Onur Mutlu, and Yale N Patt. Bottleneck identification and scheduling in multithreaded applications. In *Proceedings of the 17th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2012.

[61] José A Joao, M Aater Suleman, Onur Mutlu, and Yale N Patt. Utility-based acceleration of multithreaded applications on asymmetric cmps. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, 2013.

[62] Kurt Keutzer, Kaushik Ravindran, Nadathur Satish, and Yujia Jin. An automated exploration framework for fpga-based soft multiprocessor systems. In *2005 Third IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS'05)*, pages 273–278. IEEE, 2005.

[63] Changdae Kim and Jaehyuk Huh. Fairness-oriented os scheduling support for multicore systems. In *Proceedings of the 2016 ACM International Conference on Supercomputing (ICS)*, 2016.

[64] Changdae Kim and Jaehyuk Huh. Exploring the design space of fair scheduling supports for asymmetric multicore systems. *IEEE Transactions on Computers (TC)*, 2018.

[65] Jitendra Kumar, Richard T Mills, Forrest M Hoffman, and William W Hargrove. Parallel k-means clustering for quantitative ecoregion delineation using large data sets. *Procedia Computer Science*, 4:1602–1611, 2011.

[66] Rakesh Kumar, Dean M Tullsen, Parthasarathy Ranganathan, Norman P Jouppi, and Keith I Farkas. Single-isa heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31th Annual International Symposium on Computer Architecture (ISCA)*, 2004.

[67] Liandeng Li, Teng Yu, Wenlai Zhao, Haohuan Fu, Chenyu Wang, Li Tan, Guangwen Yang, and John Thomson. Large-scale hierarchical k-means for heterogeneous many-core supercomputers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 13. IEEE Press, 2018.

[68] Tong Li, Dan Baumberger, and Scott Hahn. Efficient and scalable multiprocessor fair scheduling using distributed weighted round-robin. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2009.

[69] Tong Li, Dan Baumberger, David A Koufaty, and Scott Hahn. Efficient operating system scheduling for performance-asymmetric multi-core architectures. In *Supercomputing, 2007. (SC). Proceedings of the 2007 ACM/IEEE Conference on*. IEEE, 2007.

[70] Weijia Li, Haohuan Fu, Le Yu, Peng Gong, Duole Feng, Congcong Li, and Nicholas Clinton. Stacked autoencoder-based deep learning for remote-sensing image classification: a case study of african land-cover mapping. *International Journal of Remote Sensing*, 37(23):5632–5646, 2016.

[71] You Li, Kaiyong Zhao, Xiaowen Chu, and Jiming Liu. Speeding up k-means algorithm by gpus. In *Computer and Information Technology*

(CIT), 2010 IEEE 10th International Conference on, pages 115–122. IEEE, 2010.

[72] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[73] Maxeler. *https://www.maxeler.com*, 2016.

[74] Sparsh Mittal. A survey of techniques for architecting and managing asymmetric multicore processors. *ACM Computing Surveys (CSUR)*, 48(3):45, 2016.

[75] Ingo Molnar. Cfs scheduler. In *Linux*, volume 2, page 36, 2007.

[76] James Newling and François Fleuret. Fast k-means with accurate bounds. In *International Conference on Machine Learning*, pages 936–944, 2016.

[77] James Newling and François Fleuret. Nested mini-batch k-means. In *Advances in Neural Information Processing Systems*, pages 1352–1360, 2016.

[78] Rina Panigrahy et al. Heuristics for vector bin packing. *research. microsoft. com*, 2011.

[79] D Pellerin. Announcing amazon ec2 f1 instances with custom fpgas hardware-accelerated computing on aws, 2016.

[80] Francesco Redaelli, M Santambrogio, Donatella Sciuto, and Seda Ogrenci Memik. Reconfiguration aware task scheduling for multi-fpga systems. *Reconfigurable Computing*, page 57, 2010.

[81] UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/datasets.html.

[82] Albert Reuther, Chansup Byun, William Arcand, David Bestor, Bill Bergeron, Matthew Hubbell, Michael Jones, Peter Michaleas, Andrew Prout, Antonio Rosa, et al. Scheduler technologies in support of high performance data analysis. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2016.

[83] Christopher J Rossbach, Yuan Yu, Jon Currey, Jean-Philippe Martin, and Dennis Fetterly. Dandelion: a compiler and runtime for heterogeneous systems. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 49–68. ACM, 2013.

[84] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[85] Juan Carlos Saez, Alexandra Fedorova, David Koufaty, and Manuel Prieto. Leveraging core specialization via os scheduling to improve performance on asymmetric multicore systems. *ACM Transactions on Computer Systems (TOCS)*, 30(2):6, 2012.

[86] Michel P Schellekens. The correspondence between partial metrics and semivaluations. *TCS*, 315(1):135–149, 2004.

[87] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In *European Conference on Parallel Processing*, pages 296–310. Springer, 2000.

[88] Malte Schwarzkopf, Andy Konwinski, Michael Abd-El-Malek, and John Wilkes. Omega: flexible, scalable schedulers for large compute clusters. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 351–364. ACM, 2013.

[89] Volker Seeker, Pavlos Petoumenos, Hugh Leather, and Björn Franke. Measuring qoe of interactive workloads and characterising frequency governors on mobile devices. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pages 61–70. IEEE, 2014.

[90] Xiao-Bo Shen, Weiwei Liu, Ivor W Tsang, Fumin Shen, and Quan-Sen Sun. Compressed k-means for large-scale clustering. In *AAAI*, pages 2527–2533, 2017.

[91] Rui Shi, Huamin Feng, Tat-Seng Chua, and Chin-Hui Lee. An adaptive image content representation and segmentation approach to automatic image annotation. In *International conference on image and video retrieval*, pages 545–554. Springer, 2004.

[92] Gabriel Southern and Jose Renau. Analysis of parsec workload scalability. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on*. IEEE, 2016.

[93] Michael Steinbach, George Karypis, Vipin Kumar, et al. A comparison of document clustering techniques. In *KDD workshop on text mining*, volume 400, pages 525–526. Boston, 2000.

[94] Mark Stillwell et al. Dynamic fractional resource scheduling versus batch scheduling. *IEEE TPDS*, 23(3):521–529, 2012.

[95] Jinwoo Suh, Dong-In Kang, and Stephen P Crago. A communication scheduling algorithm for multi-fpga systems. In *Proceedings 2000 IEEE Symposium on Field-Programmable Custom Computing Machines (Cat. No. PR00871)*, pages 299–300. IEEE, 2000.

[96] M Aater Suleman, Onur Mutlu, Moinuddin K Qureshi, and Yale N Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th international Conference on Architectural Support for Programming Languages and Operating systems (ASPLOS)*, 2009.

[97] Haluk Topcuoglu et al. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE TPDS*, 13(3):260–274, 2002.

[98] Leonardo Torok, Panos Liatsis, Jos Viterbo, Aura Conci, et al. k-ms. *Pattern Recognition*, 66(C):392–403, 2017.

[99] Kenzo Van Craeynest, Shoaib Akram, Wim Heirman, Aamer Jaleel, and Lieven Eeckhout. Fairness-aware scheduling on single-isa heterogeneous multi-cores. In *Proceedings of the 22nd international conference on Parallel Architectures and Compilation Techniques (PACT)*, 2013.

[100] Kenzo Van Craeynest, Aamer Jaleel, Lieven Eeckhout, Paolo Narvaez, and Joel Emer. Scheduling heterogeneous multi-cores through performance impact estimation (pie). In *Proceedings of the 39th Annual International Symposium on Computer Architecture (ISCA)*, 2012.

[101] Anouk Van Laer, Timothy Jones, and Philip M Watts. Full system simulation of optically interconnected chip multiprocessors using gem5.

In *Optical Fiber Communication Conference*, pages OTh1A–2. Optical Society of America, 2013.

[102] Sean Wallace, Xu Yang, Venkatram Vishwanath, William E Allcock, Susan Coghlan, Michael E Papka, and Zhiling Lan. A data driven scheduling approach for power management on hpc systems. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 56. IEEE Press, 2016.

[103] Shicai Wang, Ioannis Pandis, David Johnson, Ibrahim Emam, Florian Guitton, Axel Oehmichen, and Yike Guo. Optimising parallel r correlation matrix calculations on gene expression data using mapreduce. *BMC bioinformatics*, 15(1):351, 2014.

[104] Xiaodong Wang and José F Martínez. Rebudget: Trading off efficiency vs. fairness in market-based multicore resource allocation via runtime budget reassignment. In *Proceedings of the 21th international Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2016.

[105] Ian H Witten, Eibe Frank, Mark A Hall, and Christopher J Pal. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[106] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The splash-2 programs: Characterization and methodological considerations. In *Proceedings of the 22th Annual International Symposium on Computer Architecture (ISCA)*, 1995.

[107] Andrew Chi-Chih Yao. New algorithms for bin packing. *Journal of the ACM (JACM)*, 27(2):207–227, 1980.

[108] Andy B Yoo, Morris A Jette, and Mark Grondona. Slurm: Simple linux utility for resource management. In *Workshop on Job Scheduling Strategies for Parallel Processing*, pages 44–60. Springer, 2003.

[109] Teng Yu, Bo Feng, Mark Stillwell, José Gabriel F Coutinho, Wenlai Zhao, Shuang Liang, Wayne Luk, Alexander L Wolf, and Yuchun Ma. Relation-oriented resource allocation for multi-accelerator systems. In *Application-Specific Systems, Architectures and Processors*

(ASAP), 2016. Proceedings. The IEEE International Conference on, pages 243–244. IEEE, 2016.

[110] Teng Yu, Bo Feng, Mark Stillwell, Liucheng Guo, Yuchun Ma, and John Donald Thomson. Lattice-based scheduling for multi-fpga systems. In *Proceedings of the International Conference on Field-Programmable Technology 2018, Naha, Okinawa, Japan*. IEEE Press, 2018.

[111] Teng Yu, Pavlos Petoumenos, Vladimir Janjic, Hugh Leather, and John Thomson. Colab: A collaborative multi-factor scheduler for asymmetric multicore processors. In *Proceedings of the 2020 International Symposium on Code Generation and Optimization (CGO)*, 2020.

[112] Teng Yu, Pavlos Petoumenos, Vladimir Janjic, Mingcan Zhu, Hugh Leather, and John Thomson. Poster: A collaborative multi-factor scheduler for asymmetric multicore processors. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 487–488. IEEE, 2019.

[113] Teng Yu, Wenlai Zhao, Pan Liu, Vladimir Janjic, Xiaohan Yan, Shicai Wang, Haohuan Fu, Guangwen Yang, and John Thomson. Large-scale automatic k-means clustering for heterogeneous many-core supercomputer. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*.

[114] Seyed Majid Zahedi, Qiuyun Llull, and Benjamin C Lee. Amdahlś law in the datacenter era: A market for fair processor allocation. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018.

[115] Mario Zechner and Michael Granitzer. Accelerating k-means on the graphics processor via cuda. In *Intensive Applications and Services, 2009. INTENSIVE'09. First International Conference on*, pages 7–15. IEEE, 2009.

[116] Dengsheng Zhang, Md Monirul Islam, and Guojun Lu. A review on automatic image annotation techniques. *Pattern Recognition*, 45(1):346–362, 2012.

[117] Wenlai Zhao, Haohuan Fu, Jiarui Fang, Weijie Zheng, Lin Gan, and Guangwen Yang. Optimizing convolutional neural networks on the sunway taihulight supercomputer. *ACM Transactions on Architecture and Code Optimization (TACO)*, 15(1):13, 2018.

[118] Wenlai Zhao, Haohuan Fu, Wayne Luk, Teng Yu, Shaojun Wang, Bo Feng, Yuchun Ma, and Guangwen Yang. F-cnn: An fpga-based framework for training convolutional neural networks. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 107–114. IEEE, 2016.

[119] Xia Zhao, Zhiying Wang, and Lieven Eeckhout. Classification-driven search for effective sm partitioning in multitasking gpus. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 65–75. ACM, 2018.