Karlsruhe Institute of Technology

Co-Scheduling of Multiple Processes in Heterogeneous Systems

Masterarbeit
von

B. Sc. Meister Rados

an der Fakultät für Informatik

Tag der Anmeldung: 22. April 2019
Tag der Fertigstellung: 21. Oktober 2019

Aufgabensteller:
Prof. Dr. rer. nat. Anon.

Betreuer:
M. Sc. Anon.

Ich versichere hiermit wahrheitsgemäß, die Arbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe selbständig angefertigt, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderung entnommen wurde.

Karlsruhe, den 22. Oktober 2019

Meister Rados

Abstract

Modern high performance computing systems provide an increasing amount of on-node concurrency. However, not all applications profit from the high degree of parallelism, leaving parts of the system unused. Since operating systems nowadays allow multiple users to execute processes on the same machine simultaneously, a co-schedule between the user's processes can be consulted to remedy the idle hardware issue. This thesis introduces customized lightweight data structures and management mechanisms thereof for enabling co-scheduling of multiple processes' tasks. The data structures are accessible by any application in the system, offering great extensibility. Integration of the features in the runtime system "HALadapt" and implementation of multiple scheduling mechanisms along with suitable benchmarks show possible speedups of up to 136.22%, while significantly increasing hardware load by making use of idle devices in the system in a heterogeneous scenario. When computing on homogeneous execution hardware, increased fairness between tasks and improved hardware usage is enabled.

Contents

1 Introduction 1
1.1 Motivation 1
1.2 Goals of this Thesis 2
1.3 Thesis' Structure 3

2 Fundamentals 5
2.1 Moores' Law 5
2.2 Dennard Scaling 7
2.3 Amdahl's Law 8
2.4 Terminology 10
2.5 Problem Statement 12
2.5.1 Exemplary Scenario 12
2.6 Software 14
2.6.1 Programming Parallel Applications 15
2.6.2 Runtime System HALadapt 15
2.6.3 Shared Memory 16

3 State of the Art 17
3.1 Runtime Systems 17
3.2 Inter-Process Communication 18
3.3 Co-Scheduling Algorithms 19
3.4 Determination of Co-Scheduling 20

4 Approach 21
4.1 Determination of Co-Scheduling 21
4.2 Co-Scheduling Information Management 22
4.2.1 The Shared Memory: Implicit Application Independent Communication Interface 22
4.2.2 Data Structure Co-Scheduling Information Management File . . . 23
4.2.3 Data Structure Co-Scheduling Information 24
4.3 Co-Scheduling Flow 24
4.4 Co-Scheduling Algorithms 25
4.4.1 Round Based Greedy Algorithm 25
4.4.2 Heuristic Evolutionary Algorithm 27

4.4.3 Random Mapping Algorithms . . . 27
4.4.4 Brute Force Algorithm . . . 28

5 Implementation 29
5.1 Determination of Co-Scheduling . . . 29
5.1.1 Configurability . . . 29
5.1.2 Always . . . 30
5.1.3 Hardware Contention . . . 30
5.1.4 Speedup . . . 30
5.1.5 Waiting Time Amortization . . . 31
5.2 Inter-Process Communication . . . 32
5.2.1 Reading other processes' memory . . . 32
5.2.2 Shared Memory . . . 34
5.3 Data Structures . . . 36
5.3.1 Data Structure "Co-Scheduling Information Management File" . . . 36
5.3.2 Data Structure "Co-Scheduling Information" . . . 39
5.4 The Co-Scheduler . . . 41
5.4.1 Scheduling Flow . . . 41
5.4.2 Configurable Priority . . . 44
5.4.3 Stickiness . . . 46
5.5 Random Based Scheduling Algorithms . . . 46
5.5.1 Simple Random Mapping Algorithm . . . 46
5.5.2 Enhanced Random Co-Scheduler . . . 47
5.6 Integration in HALadapt . . . 47
5.6.1 Activation of Co-Scheduling Features . . . 48
5.6.2 Additionally Required Mechanisms . . . 48
5.6.3 Integration of the Shared Memory Features . . . 50
5.6.4 Information Needed by the Co-Scheduler . . . 51
5.6.5 Necessary Amendments . . . 51

6 Evaluation 53
6.1 Experimental Setup . . . 53
6.2 Benchmarks . . . 53
6.2.1 Mandelbrot Set . . . 54
6.2.2 Particle Filter . . . 55
6.2.3 Prime Stress . . . 55
6.3 Improvements for Computing on Heterogeneous Hardware . . . 55
6.3.1 Scenario Description . . . 56
6.3.2 Speedup . . . 56
6.3.3 Increased Hardware Load . . . 57
6.4 Improvements for Computing on Homogeneous Hardware . . . 58
6.4.1 Scenario Description . . . 59

6.4.2 Increased Fairness 59
6.5 Runtime Overhead 60
6.5.1 Runtime Overhead of the Co-Scheduling Determination Strategies 62
6.5.2 Runtime Overhead of the Co-Schedulers 62
6.6 Memory Overhead 64
6.6.1 Memory Overhead Caused by the Co-Scheduling Information Management File 66
6.6.2 Memory Overhead Caused by the Co-Scheduling Information . . 66
6.6.3 Overall Memory Overhead 67

7 Summary **69**
7.1 Co-Scheduling of Multiple Processes in Heterogeneous Systems 69
7.2 Goals Reached 69
7.3 Outlook 70

List of Tables

2.1 Dennard Scaling . . . 7
2.2 Failure of Dennard Scaling . . . 8
2.3 Exemplary Execution Times . . . 14

5.1 Header of the Co-Scheduling Information Management File Data Structure 38
5.2 Payload of the Co-Scheduling Information Management File . . . 39
5.3 Data Structure the Co-Scheduling Information Header . . . 40
5.4 Data Structure the Co-Scheduling Information Payload . . . 41

6.1 Quadratic Regression Results for the Priority Definitions' Runtime Overhead . . . 63
6.2 Quadratic Regression Results for the Plain Random Mapping Runtime Overhead . . . 64
6.3 Memory Requirement of Co-Scheduling Information Management File (CSIMF)'s Header . . . 66
6.4 Memory Requirement of CSIMF's Payload . . . 67
6.5 Memory Requirement of Co-Scheduling Information (CSI)'s Header . . . 68
6.6 Memory Requirement of CSI's Payload . . . 68

List of Figures

2.1 Moore's Law 6
2.2 Amdahl's Law 9
2.3 Visualized Execution Flow Annotated with Terminology of this Thesis . . 11
2.4 Example Schedule without Process Co-Scheduling 14
2.5 Example Schedule with Process Co-Scheduling 15
2.6 Scheme of the Shared Memory Concept 16

4.1 Data Structures in Shared Memory 23
4.2 The Co-Scheduling Flow 26

5.1 CSI Files in the Shared Memory 42
5.2 Flow Diagram of the Round Based Co-Scheduler 45

6.1 The Mandelbrot Set 54
6.2 Speedup Enabled by Co-Scheduling Multiple Processes 57
6.3 Allocation Mapping of Scenario 1 without Co-Scheduling 58
6.4 Allocation Mapping of Scenario 1 with Co-Scheduling 58
6.5 Improved Fairness by Co-Scheduling Multiple Processes 60
6.6 Allocation Mapping of Scenario 2 without Co-Scheduling 61
6.7 Allocation Mapping of Scenario 2 with Co-Scheduling 61
6.8 Runtime Overhead of Co-Scheduling Determination Strategies 62
6.9 Runtime Overhead of the Round Based Scheduler's Priority Definitions . 64
6.10 Runtime Overhead of the Enhanced Random Based Scheduler 65
6.11 Runtime Overhead of the Plain Random Based Scheduler 65

Listings

5.1 Configuration of the Co-Scheduling Determination in the File `local.mk` 30
5.2 Configurable Co-Scheduling Determination 31
5.3 Configuration of the Priority Defenition in File `local.mk` 46
5.4 Looping Mechanism to Enable Any Number of Occurring Co-Schedulings 49

6.1 Script for Evaluating the First Scenario 57
6.2 Script for Evaluating the Second Scenario 60

1 Introduction

This chapter serves for the reader as an introduction to the topic of this thesis. The first section gives a motivation for resolving the problem that is subject of this thesis, highlights the drawbacks of the current situation and presents an approach to solve it. The second section gives an overview of the goals of this thesis. The third section describes the structure of the following work.

1.1 Motivation

The steadily growing demand for more computational capacity caused a shift in hardware design. High Performance Computing (HPC) systems utilize manycore processors and specialized hardware accelerators to upkeep meeting their performance requirements. These systems are called “heterogeneous systems” [1]. However, this system configuration arises the need for congruent software properties, ie. the running program needs to inherit the property to scale well with the available degree of parallelism. Otherwise, parts of the system might be left idle at runtime, resulting in inefficient hardware usage. In order to achieve higher efficiency, multiple processes can be executed on the same system node at the same time. This performs especially well, if the running processes have mutually exclusive hardware requirements. However, this approach arises need for a mechanism that manages execution of two independent processes on the same hardware, called a “co-scheduling” of multiple processes. It is in the interest of HPC system operators to find the most efficient job schedule for their system, since it is a promising way to minimize operating costs by maximizing overall throughput at a time.

Running applications simultaneously on the same system without considering its current state, might arise various issues that could be prevented by a reasoned co-scheduling. In case multiple applications in sum instantiate more threads than the underlying hardware offers, the execution time on one core will be shared. On top of that, the processes will inevitably begin to compete for cache lines, i.e. replacing each other’s data in the cache at runtime. Problems like cache thrashing, thread starvation or threads failing to receive sufficient amount of cache needed to produce considerable throughput may arise [2] [3]. In sum, this can lead to a significantly higher cache miss rate, hence a reduction of the

Instructions per Cycle (IPC), causing an overall higher application execution time [4].
In case multiple applications compete for the same hardware accelerator, eg. Graphics Processing Unit (GPU), Intel Xeon Phi or Field Programmable Gate Array (FPGA), the behavior depends on the hardware driver or the Application Programming Interface (API) [5]. They might be scheduled, implying a serial execution equal to a First In, First Out (FIFO) schedule, or share the accelerator. However, since it is up to the driver to determine the behavior, it is hard to predict the actual runtime. In either case, the Central Processing Unit (CPU) remains uninvolved in the meantime, due to the inefficient schedule.

Task based runtime systems are suitable for tackling these problems. Incoming processes register their tasks in the runtime system and provide multiple implementations which target different hardware accelerators using various APIs (see 2.6.1). This allows for flexible execution configurations and expands the solution space for efficient co-schedules. Since new tasks can unpredictably arise over time, it is reasonable to make use of waiting queues for every processing unit. Tasks store information about eg. for how long it will occupy the processing unit in these queues such that other tasks can be scheduled with regard to the state of the hardware. Storing information about alternative executions on different hardware configurations together with the respective execution time allows for scheduling diversity. The runtime system can use this information to calculate a useful co-scheduling of all processes. This mechanism will be subject of this thesis.

1.2 Goals of this Thesis

This section gives a brief overview of this thesis' goals. The approach to fulfill each of them as will as their final accomplishment be reviewed in the following chapters of this thesis.

Goal of this thesis is to implement a mechanism to find a co-schedule of multiple processes in heterogeneous systems. This requires the processes to have the ability to detect inefficient schedules and to agree with other instances on a co-schedule. This new schedule is desired to be globally efficient. Preemption , ie. interruption of running executions, is not designated to be implemented in this thesis.
The goals at a glance:

- Detect inefficient schedule of multiple processes' tasks
- Determine point in time when to co-schedule, preventing large overhead

- Enable inter-process communication
- Create data structures that contain data required for a useful co-scheduling
- Introduce mechanism that finds a co-schedule using the customized data structures
- Implement these mechanisms into the runtime system “HALadapt”
- Demonstrate enabled speed up by running suitable benchmarks

1.3 Thesis' Structure

This section provides an overview of the structure of this thesis. Every chapter will be introduced briefly and their purpose is explained.

This first chapter, Introduction, contains the motivation for this thesis, which describes the issue that is to be solved. The goals are concluded in bullet points. The second chapter, Fundamentals assures the reader to have all information necessary in order to understand the approaches made in this thesis. An historical overview of related facts is given to describe the circumstances that led over time to the problem that this thesis aims to solve. A concluding problem statement is given and emphasized by an example. The third chapter, State of the Art sheds light on scientific work done that is related to this thesis. It serves as a literature review and compares other approaches and solutions with the ones presented in this thesis. The fourth chapter, Approach contains theoretical concepts of the approach. Design decisions met that will later be implemented are justified there. The fifth chapter, Implementation shows the reader how the actual implementation of the solution to the presented problem is done. It contains technical details, source code, explanations thereof and comments. The sixth chapter, Evaluation servers as a proof of working implementation. Suited benchmarks are used to show the extent of the newly reached speedup. The last chapter, Summary serves as a conclusion of this thesis. What could have been achieved, what can be done, what can be improved is part of this chapter.

2 Fundamentals

This chapter provides background knowledge to the reader that is required to understand how the problem, that is tackled in this thesis, arose. For this, historical, electrical and physical aspects in digital computing using transistors are considered. This reaches from Moore's Law and its consequences of Dennard's Scaling and the failure thereof to Amdahl's Law. Furthermore, the terminology used in this thesis is provided for clarity before coming to the problem statement. Furthermore, explanation on the tools used for the implementation is provided. Lastly, the runtime system on which this thesis bases is introduced.

2.1 Moores' Law

With the first working silicon transistor in early 1954 [6], the way was cleared for the first integrated circuit, which was developed shortly after that in 1958 [7]. From there, a race in scaling down transistor size was unleashed. Gordon Moore, co-founder of "N M Electronics" [8] (today Intel) anticipated that transistor's size scaling will follow a trend: Complementary Metal-Oxide-Semiconductor (CMOS) Transistors will halve about every 18 in size, leading to twice as many components on one single integrated circuit [9]. He amended his prediction later in 1975, saying the transistor count will double every two years, instead of every 18 months [10]. This observation-derived prediction is today known as "Moore's Law", without actually being a law. Figure 2.1 depicts the actual transistor count (y-axis) of integrated circuits released over the last decades (x-axis). The figure also shows a straight line: Moore's amended prediction from 1975. It is astounding how accurately this prediction can be used to predict the trend of future transistor count per integrated circuit. There were a few other researchers predicting development of other aspects of integrated circuits back in 1965. In contrast to Moore's prediction, they were not accurate enough to attract attention, or simply wrong [11].

In summary, Moore's Law implies that the component count per integrated circuit will rise exponentially over time. Nevertheless, this prediction is of theoretical nature and has no upper bound. Clearly, there is to be an end to transistor's size shrinking in reality. The

Figure 2.1: Moore's Law. [1]

fundamental barrier of transistor size is the size of an atom [12]. However, researchers face new problems even before such dimensions are even reached. Quantum effects cause dynamic threshold voltages and oscillations of the transistor's transconductance [13]. This causes unpredictable behavior complicates keeping control over the transistor. Another factor causing Moore's Law to falter is the increasing cost for producing smaller transistors. Semiconductor companies have to build a new wafer factory for every newly introduced technology node. A cost of about USD 20 billion is estimated for a 3nm wafer factory. All companies that can't afford this investment loose their market strength. From their point of view, Moore's Law can be seen as an economic disaster. Companies that want to sustain their market strength need to invest their revenues in such a factory, ending up with a no-win situation [14].

[1]Source: By Wgsimon , CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=15193542

Device Count	S^2
Device Frequency	S
Capacitive Device Power	$\frac{1}{S}$
Voltage	$\frac{1}{S^2}$
Power Density	1

Table 2.1: Dennard Scaling as observed before leakage currents became crucial.

2.2 Dennard Scaling

The Term “Dennard Scaling” describes a property of integrated circuits. It implies that for a given scaling factor S, the device count on one integrated circuit scales up by S^2, which is related to Moore’s Law. The frequency can be scaled up by a factor of S, because smaller transistors can change their on/off-state faster and have shorter signaling paths. Smaller transistors and shorter traces have a lower capacity, which scales down by factor $\frac{1}{S}$. This also means that lower voltage is sufficient to switch these capacities at the same speed, allowing voltage to be scaled down by factor $\frac{1}{S^2}$ [15].
This observation applied quite well, before transistor’s size scaled down to dimensions where leakage currents became crucial. As donated in equation (2.1), the overall power consumption of an integrated circuit is composed by multiple terms. Until leakage currents needed to be considered, it was sufficient to pay regard to the power consumed by changing charges in the capacitance of the circuit in order to estimate the overall power consumption of an integrated circuit. Then, constant power density is derived by plugging Dennard’s Scaling from table 2.1 into the factors composing $P_{switching\ capacitance}$: C_L (the cumulative parasitic capacitance) is substituted with $\frac{1}{S}$, voltage with $\frac{1}{S^2}$, N with S^2 and f with S. Expanding the product gives a constant as the result [16].

$$P_{avg} = P_{switching\ capacitance} + P_{short\ circuit} + P_{leakage} + P_{static} \tag{2.1}$$

where

$$P_{switching\ capacitance} = \frac{1}{2} \cdot C_L \cdot {V_{dd}}^2 \cdot N \cdot f \tag{2.2}$$

$$P_{short\ circuit} = K \cdot (V_{dd} - 2 \cdot V_T)^3 \cdot \tau \cdot N \cdot f \tag{2.3}$$

$$P_{leakage} = (I_{subthreshold} + I_{oxide} + I_{diode}) \cdot V_{dd} \tag{2.4}$$

$$P_{static} = 0 \text{ (in CMOS circuits)} \tag{2.5}$$

However, as transistors shrunk below sizes 100nm, electrons increasingly began to traverse the gate oxide, resulting in raising leakage currents [17]. In order to cope with that

Device Count	S^2
Device Frequency	S
Capacitive Device Power	$\frac{1}{S}$
Voltage	≈ 1
Power Density	S^2

Table 2.2: Actual power density since voltage no longer scales down.

issue, higher voltage can be applied to the transistor gate to close the transistor channel more firmly. This solution has a downside: the applied voltage can no longer be scaled down, as donated in table 2.2. This means the failure of Dennard Scaling.

Having a polynomial growth in power density leads to problems in heat dissipation. Since I_{oxide} increases with higher temperatures, even more heat is generated. Temperature runaway is the consequence, eventually resulting in catastrophic failure [18]. This thermal problem also caused a stop in frequency scaling, which had been the best way to improve performance until then.
Consequently, in order to keep up with the increasing demand of computational performance, parallel architectures are consulted. Thanks to Moore's Law, which has not (yet) failed, manufacturers are enabled to place more, smaller transistors on the same area, allowing duplication of execution pipelines per chip which can process in parallel. Such architectures increase the Instruction Level Parallelism (ILP) and therefore scale up performance without increasing frequency. In a nutshell, parallelism is the modern way to achieve higher performances without drastically increasing power density. To deliver considerable computing capabilities, HPC system nodes make full use of parallel architectures by being equipped with multicore or manycore CPUs and specialized accelerators like GPUs, Xeon Phis, Application Specific Integrated Circiut (ASIC)s or FPGAs [19].

2.3 Amdahl's Law

Instructions of a computational problem P can be classified into two groups: those parts, that profit from parallelism and those, which do not benefit from it. Even embarrassingly parallel problems [20] suffer from for example initialization overhead or Input / Output (I/O) accesses that can not be executed in parallel. Consider T to be the total execution time of P and p to be the portion of P that benefits from a parallel execution. Then, T is can be written as $T = (1 - p) \cdot T + p \cdot T$. By definition, only the latter part of the sum benefits from parallelism. Let s be the factor of parallelism applied to P. Clearly,

the equation becomes $T(s) = (1-p) \cdot T + \frac{p}{s} \cdot T$. Hence, the speedup S can be calculated using the following equation:

$$S(s) = \frac{T}{T(s)} = \frac{(1-p) \cdot T + p \cdot T}{(1-p) \cdot T + \frac{p}{s} \cdot T} = \frac{1}{(1-p) + \frac{p}{s}}$$

Therefore, for big s, the equation becomes:

$$\lim_{s \to \infty} \frac{1}{(1-p) + \frac{p}{s}} = \frac{1}{(1-p)}$$

This implies that the speedup of P converges to $\frac{1}{(1-p)}$, no matter how much parallelism is used to solve P [21].

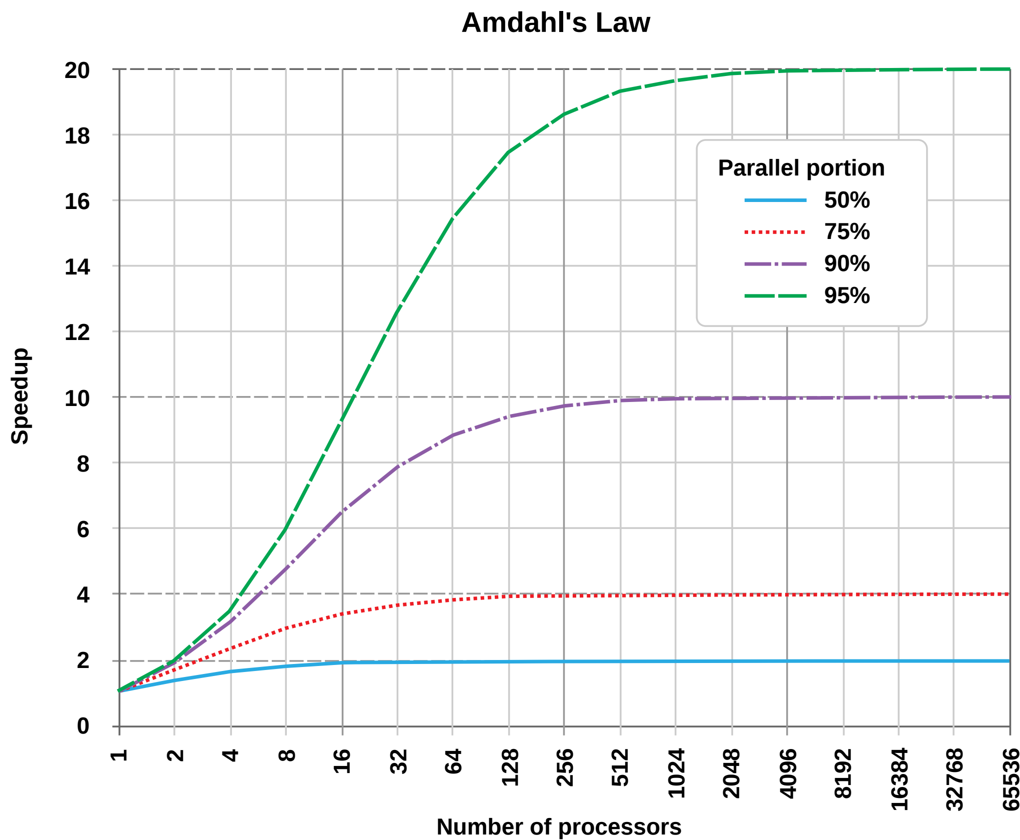

Figure 2.2: Amdahl's Law. [2]

[2] Source: By Daniels220 , CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=6678551

2.4 Terminology

This section explains the meaning of technical terms used throughout this thesis. These explanations are provided for clarity and consistency within this thesis and might not apply to any other lectures. These explanations here are in context of computing.

- **Kernel:** Code that defines the actual calculations to be done. Accepts a well defined set of input variables, which also can be empty. The Kernel is considered to be often a computationally intensive piece of code.
- **Thread:** Logical context of a kernel execution. Is managed, eg. paused, interrupted, continued by the operating system scheduler transparently for the user. Threads share the same memory if they belong to the same process. A thread is sometimes referred to as a Light-Weight Process (LWP).
- **Task:** A user provided set of at least one kernel call with a well defined set of input data, which can also be empty. There can be different kernel calls within one task, but they only differ in the targeted hardware, i.e. the same functionality is completed on the CPU, or an accelerator etc. A task needs at least one thread in which the kernel(s) reside(s). Tasks are considered to be units of computational work that the user wants to have completed.
- **Process:** One process is created for each program call. A process consists of at least one task and therefore of at least one thread. A process receives its own piece of memory which its threads are allowed to access. This memory is called “local memory”.
- **Program:** Sequence of instructions, given in source code. May include creation of arbitrary many tasks or direct kernel calls. Accepts input data from the user or another program, which is then passed to tasks or to kernels directly. Manages data flow and presents computational output to the user or other programs.
- **Library:** Set of features that the programmer can call within his program.
- **Runtime System:** A library that is designed to manage task execution. May abstract underlying hardware to ease the programming flow. Processes can submit their tasks to a runtime system which handles routines like task execution, memory transfers, scheduling, etc.
- **Application:** A program call initiated by the user. The input data can be read from the keyboard (so called “standard in”, or “stdin”) or may originate from a data file on the hard drive.
- **Co-Scheduling:** Procedure to find an execution sequence of tasks that belong to multiple processes.

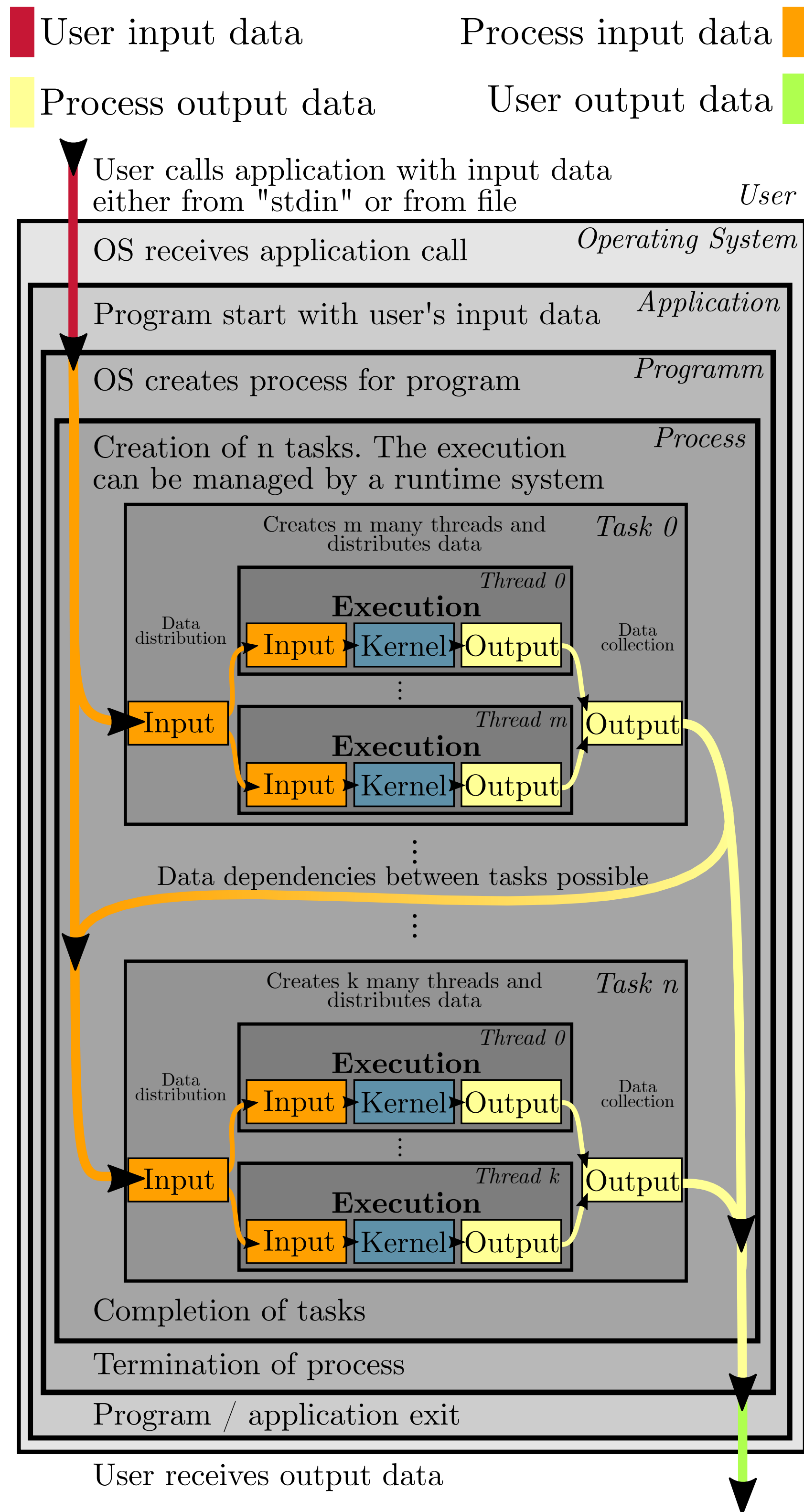

Figure 2.3: Visualized execution flow annotated with terminology of this thesis

2.5 Problem Statement

HPC systems nowadays provide a high degree of parallelism in order to achieve high performance. However, as pointed out in section 2.3, computational problems can not benefit from the provided parallelism. In order to maximize system throughput, multiple processes are executed on the same node to utilize all available hardware. This practice requires the processes to be co-scheduled such that a desired optimization goal can be achieved. This procedure may target various aspects like system throughput, energy efficiency, total system load etc.

If a HPC runtime system is not capable of co-scheduling applications, inefficient hardware usage is very likely to occur. Imagine a scenario: a user U_1 creates an instance of a runtime system at a point τ_1 in time. This instance, or process, submits 5 tasks into the runtime system which is not capable of co-scheduling tasks of multiple processes. The tasks could run on different execution devices, but since all available accelerators are idle, the scheduler decides to enqueue all tasks in the execution device providing the highest throughput.
At another instance in time, $\tau_2 > \tau_1$, a different user U_2 creates an instance of the runtime system on the same machine and enqueues 5 tasks that can only run on the accelerator that has been allocated by the tasks of U_1. Without capability of co-scheduling tasks of multiple processes, the submitted tasks by user U_2 either have to wait until the tasks of U_1 are completed or have to share the same device, slowing both executions down.
If the runtime system is capable of co-scheduling multiple processes, the tasks of U_1 that are not yet running could be transferred to execution devices for which they also offer an implementation, unblocking the execution of U_2's tasks. The slowdown of multiple processes running together in the same system is referred to as "co-run penalty".

2.5.1 Exemplary Scenario

Let S be a heterogeneous system made up of 8 CPUs and one GPU. Let P_i with $i \in \{1, 2\}$ be processes. Further, let $P_{i\ j}$ be a task in process P_i, with $j \in \{1, ..., 5\}$ and let $T(P_{i\ j}, n)$ be the execution time of task j in P_i, where n is the number of CPU threads used for executing $P_{i\ j}$. The unit is milliseconds and $n \in \{1, ..., 8\}$. If no value for n is specified, execution on the GPU is assumed. Let $S(P_{i\ \tau}, n)$ be the speedup of $T(P_{i\ j}, n)$ compared to the execution time $T(P_{i\ j}, 1)$. If no n is specified, GPU execution is assumed once again. Let T_{total} be the time it takes for all processes P_i to have their tasks $P_{i\ j}$ executed.
Let P_1 and P_2 be two processes that will be running in S. Each of these processes consist of 5 tasks $P_{i\{1,...,5\}}$ that are to be executed. $P_{2\ j}$ do not offer an implementation for the

GPU. Values for $n, T(P_{i\ j}, n)$ and $S(P_{i\ ju}, n)$ are given in table 2.3.

Let S have no running tasks initially; all hardware devices are idle. Assume P_1 being the first process to submit its tasks $P_1\ j$into the runtime system. P_1 will query the status of the available execution units, and, in order to minimize its execution time, allocate all 8 CPUs for processing its tasks $P_{1\ j}$ since this provides the highest throughput.// P_2 submits its tasks $P_2\ j$ afterwards. Like P_1, P_2 queries the status of the system and thereby acquires the information that all 8 CPUs are in use currently. Consequently, since the tasks $P_2\ j$ do not support an execution using the GPU, it has no other options than to either wait, or share the CPUs. If P_2 waits until the execution of $P_{1\ j}$ are completed and afterwards makes use of all 8 CPUs, T_{total} will be $5 \cdot 115.801ms + 5 \cdot 328.102ms = 2219.515ms$. The GPU is left idle in the meantime. Hence, the total system load is either $8/9 \approx 88\%$, or, if the CPU is seen as a monolithic device, $1/2 = 50\%$.
P_2 also has the option to share the 8 CPUs with $P_{1\ j}$. Assuming sharing one CPU among two processes causes a doubled execution time for each task, the 8 CPUs are shared for $2 \cdot 5 \cdot 115.801ms = 1158.01ms$ until $P_1\ j$ are completed. $P_2\ j$ is running for another $5 \cdot 328.102ms - 5 \cdot 115.801ms = 1061.505ms$. Therefore, T_{total} = $1158.01ms + 1061.504ms = 2219.515ms$ ignoring cache issues that might cause additional co-run penalty.
If S had the ability to co-schedule $P_{1\ j}$ and $P_{2\ j}$, shorter execution times could be achieved, since all tasks that are not running can be transferred to other execution devices in order to unblock waiting tasks. Since we assume in this example, that P_1 has submitted its tasks $P_{1\ j}$ before P_2 submitted $P_{2\ j}$, at least task $P_{1\ 1}$ is already running on 8 CPUs. This task can not be moved over to the GPU without preemption in order to unblock $P_{2\ 1}$. However, the other tasks $P_{1\ \{2,3,4,5\}}$ can be moved to the GPU, allowing earlier execution of $P_{2\ j}$. That means that $P_{2\ j}$ can begin their execution on 8 threads, as soon as $P_{1\ 1}$ finishes the execution and its succeeding tasks are then transferred to the GPU. $P_{2\ j}$ ideally can then start their execution on 8 threads after 115.801ms. $P_{1\ \{2,3,4,5\}}$ then finish their execution after $1 \cdot 115.801ms + 4 \cdot 201.337ms = 317.139ms$. $P_{2\ j}$ finish their execution after $1 \cdot 115.801ms + 4 \cdot 328.102ms = 1756.311ms$ which yields a speedup compared to the non-co-scheduled execution of $\frac{2219.515ms}{1756.311ms} = 126.37\%$.

Besides the overall speedup of $P_{1\ j}$ and $P_{2\ j}$, the system's load is increased from $\frac{8}{9} \approx 88\%$ to $\frac{8}{9} \cdot \frac{115.801ms}{1756.311ms} + 1 \cdot \frac{4 \cdot 201.337ms}{1756.311ms} + \frac{8}{9} \cdot \frac{1756.311ms - (4 \cdot 201.337ms + 115.801ms)}{1756.311ms} \approx 93.98\%$
Or, if the CPU is seen as a monolithic device, the system's load is increased from $1/2 = 50\%$ to $\frac{1}{2} \cdot \frac{115.801ms}{1756.311ms} + 1 \cdot \frac{4 \cdot 201.337ms}{1756.311ms} + \frac{1}{2} \cdot \frac{1756.311ms - (4 \cdot 201.337ms + 115.801ms)}{1756.311ms} \approx 72.92\%$

Other scenarios are possible, in which it is useful to the split amount of allocated CPUs among multiple processes. Since tasks do not benefit from arbitrarily large portions of

Device	n	$T(P_{1\ \tau 1}, n)$	$S(P_{1\ \tau 1}, n)$	$T(P_{2\ \tau 2}, n)$	$S(P_{2\ \tau 2}, n)$
CPU	1	456.974	100.00%	2568.818	100.00%
CPU	2	231.780	197.15%	1285.409	199.84%
CPU	3	164.420	277.78%	858.272	299.30%
CPU	4	133.707	341.77%	645.204	398.15%
CPU	5	124.008	368.50%	517.763	496.14%
CPU	6	120.069	380.59%	433.136	593.07%
CPU	7	116.824	391.11%	372.971	688.73%
CPU	8	115.801	394.62%	328.102	782.93%
GPU	-	201.337	226.96%	-	0%

Table 2.3: Exemplary execution times for $P_{1\ \tau 1}$ and $P_{2\ \tau 2}$ in S

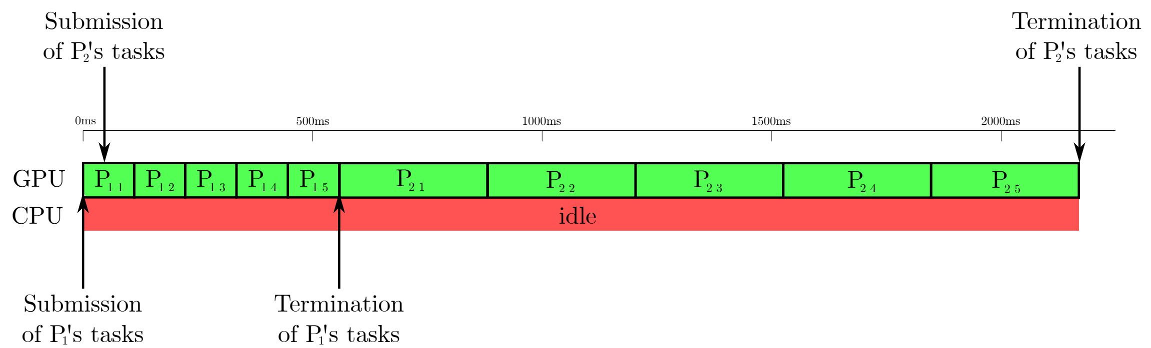

Figure 2.4: Example schedule without process co-scheduling.

parallelism, a co-scheduling can also yield a speedup for homogeneous computing systems [22].

2.6 Software

This section describes the software that is used to realize parallel programs. Also, the runtime system is introduced in which the implementation of co-scheduling is integrated. The programming language used in this thesis is C.

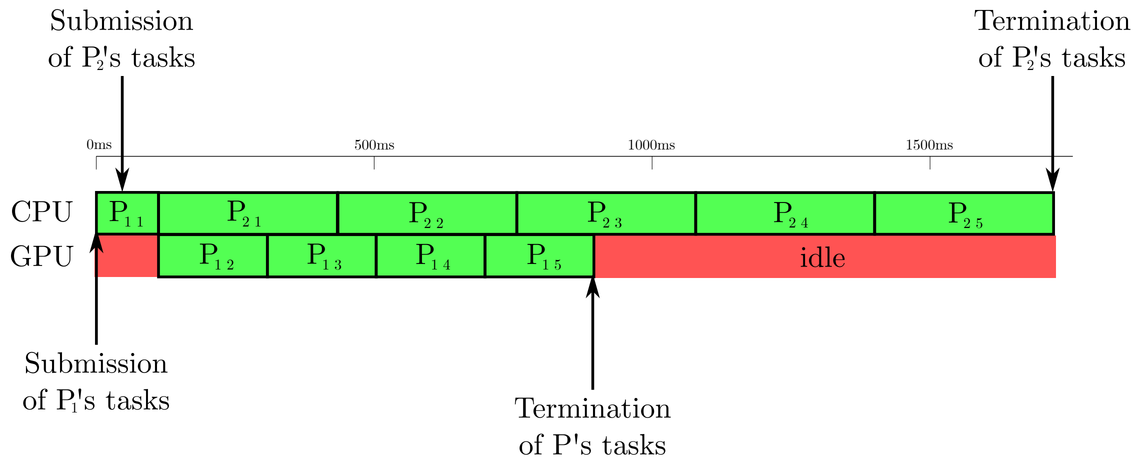

Figure 2.5: Example schedule with process co-scheduling.

2.6.1 Programming Parallel Applications

The commonly used API for making use of parallelism in a manycore CPU is Open Multi-Processing (OMP) [23]. It provides simple commands to automatically distribute computational work to all available CPU cores and bases on a fork-join model. Benchmarks that are later be used for evaluation make, among other APIs, use of OMP.
To target parallelism between computing nodes, Message Passing Interface (MPI) can be used. It is a standardized description of routines that allow message passing between multiple computing nodes [24]. However, this thesis focuses on solving problems within one single node, therefore MPI is not used or further explained.
Open Computing Language (OCL) is the solution for targeting various heterogeneous accelerators [5]. It is used throughout the evaluation to execute code on the GPU, but could also be run on any other hardware accelerator installed in the system. Besides an implementation of the benchmarks in OCL, another API, Computing Unified Device Architecture (CUDA), is used to target the GPU. CUDA is an API developed by Nvidia which enables the programmer to make use of Nvidia GPUs' computational power.

2.6.2 Runtime System HALadapt

HALadapt [25], is a runtime system developed at Karlsruhe Institute of Technology (KIT), designed to reduce the complexity in application programming for heterogeneous hardware architectures. It is written in C language and designed to run on Linux operating

system. Portability to Microsoft Windows is partly given. HALadapt allows multiple implementations for the same kernel (so called proxies) in order to target various computing architectures. HALadapt abstracts the underlying hardware and has the ability to detect which task-to-execution-device mapping provides the largest throughput. A Directed Acyclic Graph (DAG) task graph is generated from task mapping, which is then started autonomously. Mandatory memory transfers, that arise when accelerators are used for execution, are abstracted to reduce the programmer's memory management overhead. HALadapt provides waiting queues for every execution unit installed in the system in the shared memory. They are used for inter-process hardware awareness, ie. other HALadapt instances can query which devices are currently in use and take this information into account when it comes to scheduling its tasks [26].

2.6.3 Shared Memory

Co-scheduling tasks of multiple processes inherits the need for inter-process communication. Information between the processes needs to be exchanged. Since processes receive their own region in the host memory, variables in the source code can't simply be accessed by all processes. Therefore, inter-process communication can not be achieved without further effort. In order to cope with this issue, Portable Operating System Interface (POSIX) systems expose a device mounted at `/dev/shm` that can be accessed by all processes in the system. The file system is `tempfs`. As depicted in figure 2.6, the shared memory concept allows multiple processes to map the exact same region of memory into their logical address space. Effectively, accesses to that mapped region are no slower than a regular memory access into the region that has been exclusively assigned by the operating system. However, since multiple processes can now access the same physical memory addresses asynchronously, data corruption may occur. Locking mechanisms can be consulted in order to prevent this circumstance, which introduces an overhead to this approach.

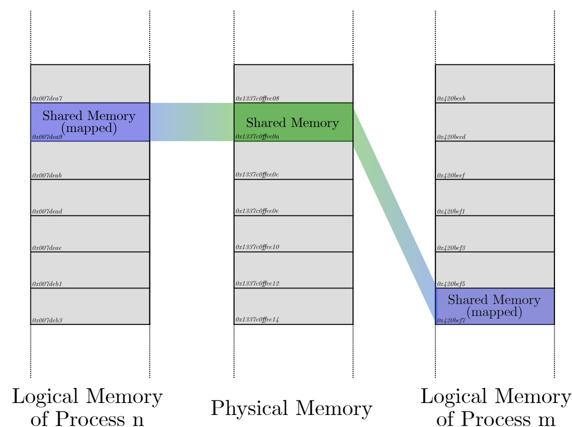

Figure 2.6: Scheme of the shared memory concept

3 State of the Art

The recently increased presence of heterogeneous computing systems and their efficient usage has made up a new field of research. This chapter presents and discusses works published in the literature that is relatable to the topic of this thesis. It includes scheduling in heterogeneous systems, also with multiple optimization targets and strategies, runtime systems for heterogeneous architectures and shared memory utilization for inter-process communication. Approaches made in these publications are assessed and, if considered useful, captured for the approach in this thesis. Differences from the presented literature to this thesis are pointed

3.1 Runtime Systems

Co-scheduling in heterogeneous computing systems requires the ability to execute submitted tasks on different execution devices. Since offloading computational load to an accelerator introduces overhead like memory management and transfers, it is advised to use a runtime system that abstracts such duties. StarPu [27] is a runtime system developed by Augonnet et al. at the university of Bordeaux. StarPU provides a high level execution model for heterogeneous computing and abstracts memory transfers from host to accelerator and vice versa. Its library based approach offers users the ability to implement the same kernel targeting multiple execution devices and accelerators. It also assures coherency between these memories. It is designed to allow the user to construct and refine own scheduling algorithms. This feature is presented to be the main subject of the work. However, StarPU does not support co-scheduling of multiple StarPU instances on one computing node.

Closely related to StarPU is HALadapt [25], a runtime system for heterogeneous computing, invented by Kicherer et al. at KIT (see section 2.6.2). Unlike StarPU, HALadapt does not predominantly focus on offering user-adjustable scheduling mechanisms. They can be added as a plugin by the user. HALadapt rather focuses on maximizing system throughput by taking measurements of past executions into account. HALadapt constructs a DAG from all submitted tasks, which also contains implicit tasks like mandatory memory transfers for the desired execution. When it comes to co-scheduling applications, HALadapt makes use of shared waiting queues for each execution device in the system. Nevertheless,

HALadapt lacks capability for re-ordering the entries in the waiting queues, and therefore can not ensure efficient co-scheduling of multiple processes. Jobs either have to wait until the waiting queues are processed and then enqueue their desired schedule, or agree on a schedule using the hardware devices that are not busy at this point[26]. This implies that only an useful schedule within the first started process can be found with HALadapt.
Another approach for seamlessly migrating tasks between hardware accelerators is clSURF [28], an OCL implementation of Open source Speeded Up Robust Feature (OpenSURF) [29]. clSURF introduces a so called work pool, which consists of multiple task queues. Enqueued tasks are equipped with meta data like estimated completion time, which is used by a scheduling algorithm to assigns the tasks to computing devices. Authors state that there is only one static scheduling mechanism, where the programmer has to decide which kernel will be executed on which device currently. A dynamic scheduler is left for future work. The so called resource management takes care for seamless migration of tasks between computing devices on demand. However, the scheduling scope of clSURF is once again at task level and no inter-process communication is realized. Therefore, clSURF is not capable of co-scheduling multiple instances of itself.

3.2 Inter-Process Communication

So far, only related work that does not support co-scheduling of multiple processes has been introduced. An implementation that supports such mechanisms is schedGPU [30]. This work is designed to co-schedule multiple processes to the same GPU while ensuring that no process is running out of memory. There are two approaches for inter-process communication presented. The first one is a client-server model where a schedGPU server manages GPU memory. Applications then act as a schedGPU client and have to request memory for execution from the schedGPU server. The second approach presented for inter-process communication is using shared memory in the host system. The authors prefer this approach since there is no single point of failure, ie. the schedGPU server. Data stored in the shared memory is: accessible GPU memory in the system, utilized portion thereof, identifiers of running processes that consume GPU memory and a queue holding all processes that requested more memory for execution that was available when being started. In contrast to the subject of this thesis, schedGPU only regards execution on the GPU and does not support migration of tasks between various accelerators in order to increase system's throughput.
Newsom et al. also presented co-scheduling at process level [22] by decomposing the processes' tasks into tasks which are then co-scheduled. Inter-process communication is achieved by using MPI. The focus of this work is on maximizing energy efficiency of HPC clusters, which differs from the subject of this thesis. Also, only execution on CPU cores is regarded. Nevertheless, job re-ordering is presented. For that, authors of [22]

introduce a simulator which evaluates all permutations of enqueued jobs and choose the execution sequence which provides the highest energy efficiency.
Another work that makes use of shared memory to co-schedule multiple processes is presented by Jiménez et al. in [31]. Like HALadapt, this work also utilizes a history based scheduling mechanism in order to predict process' waiting time when being enqueued. Besides that, a first-free scheduling is provided as an alternative. The main difference between the work by Jiménez et al. to this thesis is the support for OCL. This means that their work cannot be used in combination with accelerators like FPGAs, Xeon Phis or GPUs by the manufacturer Advanced Micro Devices (AMD).
VarySched [32] is an implementation of a process co-scheduler for heterogeneous computing architectures. The inter-process communication is achieved using Library for Accelerated Math Applications (LAMA) [33], an open source C++ library for sparse linear system solves. Offloading inter-process communication to this library omits the need for an own implementation, but binds VarySched's capabilities to those of LAMA, meaning that VarySched can only execute what LAMA is designed for, ie. solving sparse linear systems. This confinement is the main difference to this thesis that is designed to co-schedule any kind of computational job.

3.3 Co-Scheduling Algorithms

Publications like [34] and [35] by Jiang et al. proved that finding an optimal co-scheduling is NP hard if the jobs are subject to be scheduled on more than 3 CPU cores. Therefore, finding an optimal solution using brute force can, depending on the input data, take unreasonable time. For proving the problem's NP-hard property, a reduction to Multidimensional Assignment Problem (MAP) [36] is done. This proof also applies to heterogeneous computing systems, since the available accelerators can theoretically also be regarded as a regular execution device from the scheduler's point of view. Hence, only approximately optimal solutions can be found in polynomial time [1] in both scenarios, a heterogeneous computing environment, and the Chip Multiprocessor (CMP) setup regarded by Jiang et al. They present multiple approaches for finding approximately optimal solutions including Integer Programming Model (IPM) and various heuristics: a greedy algorithm, a hierarchical approach that divides the problem into smaller portions that can be solved efficiently and at last, a local optimization algorithm. The presented results reveal that the solutions found using heuristic algorithms can be close to the optimal solution. This work has been improved by Tian et al., by also regarding jobs with different lengths and job migration [37]. Nevertheless, these publications do not regard heterogeneous environments which offer accelerators that cause overheads like memory

[1] Assuming $P \neq NP$

transfers.
Another implementation of job co-scheduling is presented in [38]. In this work, effort is spent to find an optimal co-schedule of multiple processes in computing clusters. The approach made focuses on bandwidth and memory usage of the jobs. The work does not regard heterogeneous computing architectures and the proposed scheduling algorithm focuses on dividing jobs into processes which are then distributed over computing nodes so that the communication overhead is minimized. Minimizing execution time could be implied by this, but is not explicitly pursued to be achieved.

3.4 Determination of Co-Scheduling

This thesis also examines at what point a co-schedule should be found in order to achieve a higher throughput in heterogeneous computing systems. However, no work in the literature can be found that also examines this particular question. Therefore, this thesis contributes to find an answer to that question.

4 Approach

This chapter describes the approach made in this thesis to reach the goals presented in section 1.2. Approaches from related work (see chapter 3) that have been proven to be useful are inherited in this approach. Also, modifications thereof and other necessary implementations are specified here. Required customized data structures are introduced and design decisions are justified. First, the question on when to arrange a co-scheduling is regarded. After that, in section 4.2, the management of information needed to find a co-scheduling is discussed. Customized data structures are presented in this section. After that, the co-scheduling flow is described. Lastly, the approach for the actual co-scheduler is given.

4.1 Determination of Co-Scheduling

The first issue to address is the question when to co-schedule processes' tasks to execution devices. Since the co-scheduling process can introduce considerable overhead, it is useful to elaborate multiple approaches to the question if a co-schedule is useful in a given situation. The possibilities regarded in this thesis to answer this question are as follows:

- **Always:** A naive approach is to calculate a co-scheduling every time a new task is submitted by any process into the system. This ensures that no inefficient schedules may occur among all tasks that are not running (preemption is not designated in this thesis). Nevertheless, significant overhead can be expected to accompany this procedure.
- **Hardware Contention:** A more suitable idea is to calculate a co-schedule for waiting applications as soon as at least one of them is unable to allocate the desired hardware resources that promise highest throughput. What execution devices are desired by an application can be ascertained by profiling and logging execution times. This approach also ensures that the system's throughput is maximized at any point of time. Nonetheless, the overhead caused by the co-scheduling process might be greater then the thereby enabled speedup.
- **Speedup:** Another approach that avoids the aforementioned circumstance is to establish awareness for the additional cost caused by the re-scheduling as well as for

the thereby enabled speedup. With these pieces of information, a re-scheduling process can be omitted, if its cost has been detected as not amortizable by the possible speedup. The cost of a re-scheduling process can be derived experimentally and the enabled speedup can be optimistically obtained by profiling the application's performance on different execution devices. However, the actual speedup enabled through co-scheduling the processes' tasks can be only obtained by calculating the very same thing. This causes exactly the overhead whose omission was aimed for. Hence, an optimistic estimation can be pursued by assuming that all processes' tasks receive the hardware on which they have the shortest runtime.

- **Waiting Time Amortization:** A fourth approach is to re-schedule processes' tasks if all execution devices are occupied for at least the time it takes to calculate a co-scheduling. In this case, the overhead caused by calculating a co-schedule is concealed by waiting time of the afflicted tasks.

4.2 Co-Scheduling Information Management

The next concern to treat is the realization of inter-process communication. As presented in chapter 3, multiple possibilities exist. Comprising the gives facts of chapter 3, a reasonable way is to make use of the shared memory in the system. Therefore, the implementation of inter-process communication in this thesis uses the shared memory.
An easy way to co-schedule the tasks of all processes is to copy the local task data structures to the shared memory. Then, the process that initiates a the co-scheduling has access to the task data structures of all processes, allowing it to forward the information from the shared memory to its internal scheduler in order to co-schedule all processes' tasks. However, copying the task structure of HALadapt, the runtime system in which this mechanism is to be integrated, would cause an enormous overhead and need for management due to the extent of HALadapt's task data structure. This issue is further explained in section 5.2.2. Instead, customized data structures are stored in the shared memory that hold merely the data required to find a co-schedule. This comprises a management file (called CSIMF) and the actual data files (called CSI files). Details on them are presented in the following subsections.

4.2.1 The Shared Memory: Implicit Application Independent Communication Interface

As mentioned, this thesis makes use of the shared memory for inter-process communication. This

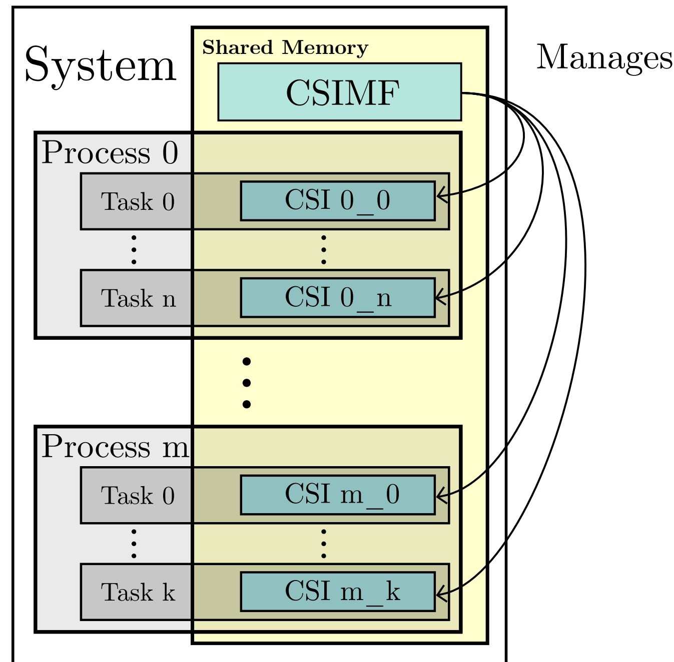

Figure 4.1: CSIMF and CSI files in the shared memory

4.2.2 Data Structure Co-Scheduling Information Management File

In order to realize the co-scheduling flow, information management and process coordination is required. These tasks are imposed on a so called “Co-Scheduling Information Management File”, in short CSIMF. This file can only exists once per node and resides in the shared memory. It has a static name so that every instance of HALadapt can easily find and access it.
The CSIMF provides mutual exclusion capabilities for itself and all CSI files respectively. This is required since multiple process might try to access or modify the data in the CSIMF or any CSI file at the same time asynchronously.
Besides the access management, the CSIMF also stores information needed for processes to find other processes’ CSI files in the shared memory. If a CSI file is generated by any process, it needs to be registered at the CSIMF. This is done by depositing an unique identifier in the CSIMF that is equal to the name of the CSI file. This ensures that every process knows what CSI files exist in the shared memory. The submitting process needs to stay aware of the submitted CSI files in order to remove them from the CSIMF if the corresponding CSI file no longer exists.

4.2.3 Data Structure Co-Scheduling Information

The CSI files hold all information needed to co-schedule the tasks of multiple processes. One CSI file is generated for each task in a process. The contained pieces of information are:

- Execution time of every possible mapping to execution devices in the system that this task supports
- Dependencies on other tasks
- Reverse dependencies, ie. the tasks that depend on this task

If a task does not provide an implementation targeting a specific accelerator available in the system, the CSI does not have an entry for this particular accelerator. Meaning that the CSI entries are populated dynamically. These stored pieces of information are used by the co-scheduler in order to find a useful co-scheduling of the processes' tasks. The presence of an CSI in the shared memory implies that the corresponding task is not running yet, and is subject to a re-scheduling. Tasks delete their CSI as soon as the execution thereof is started. Thus, the absence of a CSI implies that the task is either running or has its execution finished already.

4.3 Co-Scheduling Flow

The co-scheduling flow is depicted in figure 4.2. Incoming processes store for all of their tasks CSI files in the shared memory. They are designed to do this, even if a co-schedule is not desired. This ensures that other processes can immediately acquire the information needed for a co-scheduling, without causing overheads like inter-process signaling, by requesting already scheduled processes to store their CSIs afterwards.
After the CSIs are stored in the shared memory, the incoming process needs to determine, whether a co-scheduling is necessary or not (see 4.1).
If so, all entries from the shared memory waiting queues that are not marked as "in execution" are removed. This prevents an old schedule from being executed. Afterwards, the co-scheduler is engaged and supplied with required information from the CSIs. All tasks that have an CSI in the shared memory are implicitly waiting to be executed and therefore are subject for a co-scheduling. The co-scheduler then stores its result to the shared memory waiting queues and notifies all processes that their schedule has changed, to ensure that they will adhere to the newly assigned schedule.
From this point on, the flow is equal to the procedure in case no co-scheduling is desired. The processes execute their tasks with the assigned processing units and are removed from

the shared memory waiting queues as soon as they have finished. The corresponding CSIs are deleted at this point, too.

4.4 Co-Scheduling Algorithms

As shown in [35], the co-scheduling problem is NP hard if the number of processing units is greater than two, which is usually the case in heterogeneous computing systems. Since finding an optimal solution can take considerable amount of time with increasing magnitude of problem size, a heuristic based approach will be pursued. This section briefly describes some algorithms that can be used to find a co-scheduling.

4.4.1 Round Based Greedy Algorithm

The algorithm that is implemented in this thesis can be seen as a round based greedy scheduling algorithm for co-scheduling tasks in heterogeneous systems. Unlike the closely related Heterogeneous Earliest Finish Time (HEFT) scheduler [39], it allows to configure the criterion used to decide which task may allocate an accelerator in different situations. It can be chosen between the runtime of the task or the amount of succeeding tasks. Situations in which this criterion is applied can be either hardware contention or when free accelerators are distributed to tasks that can make use of them.
However, like all heuristic algorithms have in common, this scheduler is prone to local extrema, whilst offering a reasonable runtime for solving a NP-hard problem.
In order to make the scheduler more unique and fit the situation, some assumptions are made for finding a solution:

- **Multiple independent tasks ready for execution:** Since the scheduler is designed to co-schedule tasks of multiple processes, we can assume that there are tasks to co-schedule which do not depend on each other, as they belong to different processes.
- **Accelerators provide higher throughput:** Available heterogeneous computing hardware offers specialized execution units which serve for shorter runtime than on the CPU. The scheduler therefore treats these accelerators separately to keep their usage high.
- **Memory transfers are expensive:** The transfer of data to and from the accelerator is also taken into account and, since considerable overhead might occur, tried to be avoided. Since this assumption might be conflicting with the preceding one, the scheduler provides a feature called “stickiness”. Further details are provided in section 5.4.3.

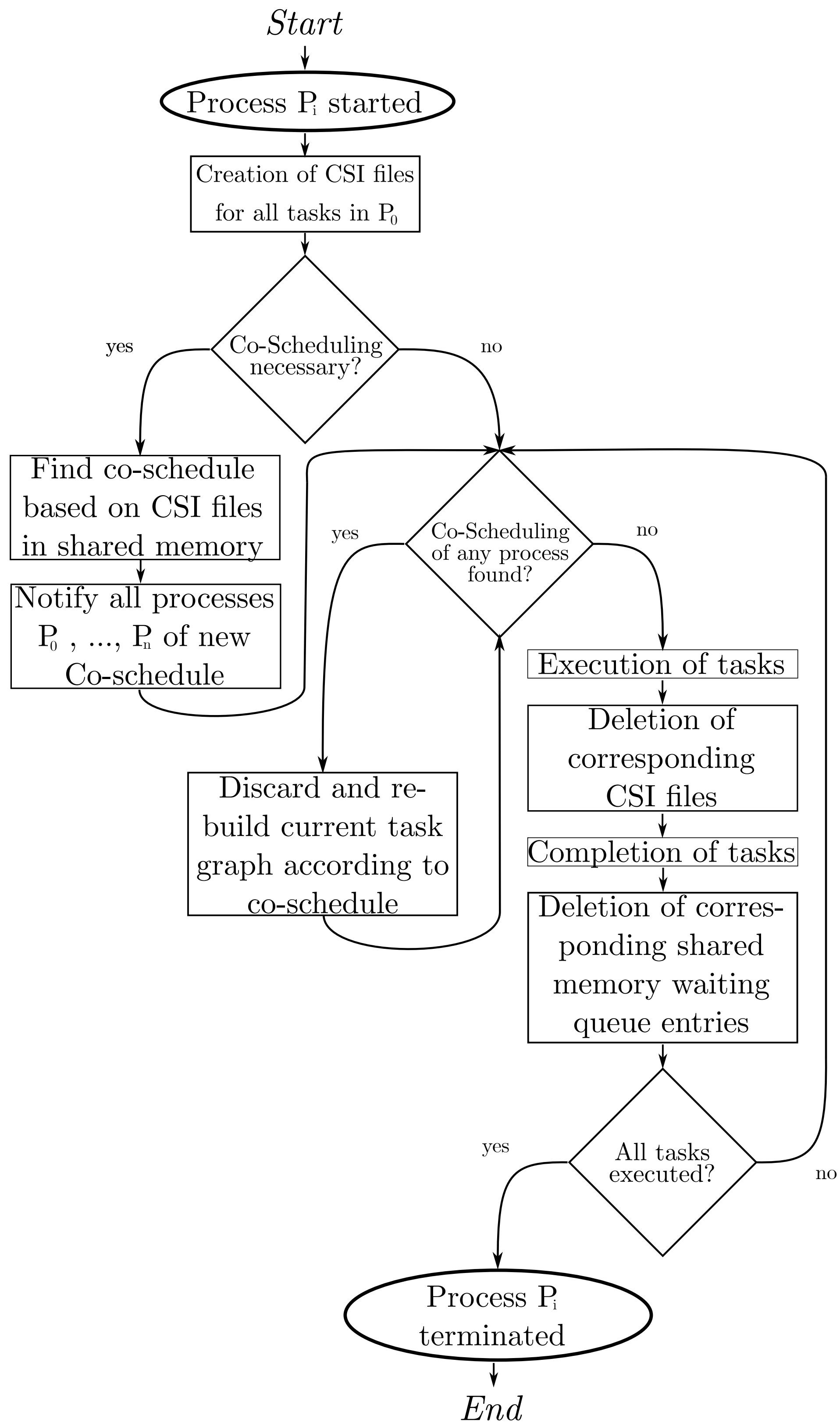

Figure 4.2: The co-scheduling flow

4.4.2 Heuristic Evolutionary Algorithm

Another approach for approximately solving NP-hard problems in reasonable time are heuristic evolutionary algorithms [40]. Similar to related iterative algorithms, like the Metropolis Algorithm or Simulated Annealing [41], evolutionary algorithms may overcome local extrema by randomly elaborating solutions which do not provide a better solution than the one started with. The main difference to Simulated Annealing is that evolutionary algorithms do not work on a single solution, they work on a set of possible solutions. This set is called a population to which three main operations can be applied iteratively:

- **Selection:** A fitness function is applied to all individuals in the current population in order to quantify their respective fitness. Based on that, individuals can be discarded for future populations. The amount or the threshold can be chosen randomly or adjusted over time. This might lead to convergence to the optimal solution over many iterations.
- **Mutation:** Individuals in the population are modified. The modification's magnitude can be chosen randomly or adjusted over time. This might lead to completely new individual's characteristics and therefore supplies the population with diversity, allowing to overcome local extrema.
- **Crossover:** New individuals are created based on the characteristics of other individuals. What parts are transferred into the next generation can be chosen randomly, as well as the amount of transferred characteristics. The thereby created individuals are referred to as "children".

4.4.3 Random Mapping Algorithms

An simple approach is to generate a random mapping from tasks to execution devices for which the task offers an implementation. This approach can find a perfect mapping in constant time. However, the probability for this is vanishingly small. A random mapping algorithm, unlike the round based greedy algorithm, does not have the disadvantage to be likely to run into a local extrema. Therefore a random mapping algorithm can not get stuck in a local optimum that leads to inability for finding the desired global one.

This thesis presents, along with the plain random mapping scheduler, another scheduling approach that bases on randomness. This scheduling algorithm referred to as "enhanced random mapping scheduler", which is aware of the available hardware in the system, as well as of the maximum amount of forward dependencies in between the tasks that are scheduled. The algorithm is allowed to distribute all hardware to the tasks as many times

as the maximum amount of forward dependencies among the tasks. The actual mappings are chosen randomly and the thereby allocated hardware is summed up. If the sum exceeds the allowed amount of allocated hardware, the mapping is discarded. The enhanced random mapping scheduler repeats this procedure for a specific amount of times and then applies the mapping found with the shortest runtime.

4.4.4 Brute Force Algorithm

A straightforward way to optimally solve a problem is to generate and compare all possible solutions. This ensures that the optimal solution is found, but is likely to produce significant overhead in terms of runtime and memory requirement for the problem sizes that are of interest. For example, there are $\sum_{i=1}^{48} \binom{48}{i} = 2^{48} - 1 = 281,474,976,710,655$ ways to execute an OMP task on a machine with 48 threads. A CPU running at 3GHz would need more than 26 hours to count to that number. The generation of that many scheduling candidates is expected to take even longer. Another concern is the memory requirement. The size of a candidate data structure in HALadapt is 160Bytes and contains 5 pointers to other required memory constructs, of which 3 are lists of variable length. Taken only the plain scheduling candidate construct into account, more than 4.19GiB of memory in the host machine are required to store all candidates of this single task.
Due to these circumstances, calculating the optimal solution is not pursued in this thesis.

5 Implementation

This Chapter describes details of the actual implementation of the enhancements of HALadapt, which comprise the determination of need for a co-scheduling in all four presented variants, the implementation of inter-process communication, the data structures managing and holding the co-scheduling information, the scheduler itself that makes use of the information in the shared memory together with its features and the integration in HALadapt.

5.1 Determination of Co-Scheduling

This section presents the implementation of the four different determination procedures for a co-scheduling need. Process initiate calculation of a process-wide co-scheduling based on the decision of the four strategies.

5.1.1 Configurability

The user can configure which of the four strategies is used to determine the co-scheduling. This is done by setting one of the flags `-DDLS_CO_SCHEDULING_STRATEGY_ALWAYS`, `-DLS_CO_SCHEDULING_STRATEGY_HARDWARE_CONTENTION`, `-DDLS_CO_SCHEDULING_STRATEGY_SPEEDUP` or `-DDLS_CO_SCHEDULING_STRATEGY_AMORTIZATION` in a local configuration file called `local.mk` in the library directory of HALadapt. Lines that are preceded by a hashtag (`#`) are ignored by the `make`, the tool that is used to pass the flags from the configuration file to the compiler. This allows the user to merely add or remove a hashtag when changing co-scheduling determination strategy. On rebuild of the library, the desired flags are passed to the C preprocessor which copies the corresponding features into the source code before it is compiled by the C compiler. The flags are not mutually exclusive and multiple can be set at a time. However, as shown in code listing 5.2, all inserted code snippets contain a `return` statement which will cause the routine exit, rendering

the rest of it to be effectively dead code.

```
201 # always do a co-scheduling if new task is submitted by any process
202 #OPT_DEFS += -DDLS_CO_SCHEDULING_STRATEGY_ALWAYS
203
204 # only do a co-scheduling if hardware contention is detected
205 OPT_DEFS += -DDLS_CO_SCHEDULING_STRATEGY_HARDWARE_CONTENTION
206
207 #co-schedule tasks if the speedup is greater than its calculation costs
208 OPT_DEFS += -DDLS_CO_SCHEDULING_STRATEGY_SPEEDUP
209
210 #co-schedule tasks if the have to wait longer than its calculation takes
211 #OPT_DEFS += -DDLS_CO_SCHEDULING_STRATEGY_AMORTIZATION
```

Listing 5.1: Configuration of the Co-Scheduling Determination in the File `local.mk`

5.1.2 Always

If the flag `-DDLS_CO_SCHEDULING_STRATEGY_ALWAYS` is passed to the compiler when building the library, the query for a co-scheduling need returns “1”, in all cases. This will lead to a co-scheduling calculation and initiation regardless of any factors that could a co-scheduling render useless. Choosing this option even excludes sanity check that would prevent a co-scheduling calculation in case there are no other processes on the node that have submitted tasks into the runtime system that could be co-scheduled. Consequently, choosing this option can result in passing the co-scheduler tasks of only one process which makes it face a situation for which its assumptions are not congruent.

5.1.3 Hardware Contention

This strategy is inserted into the code, when the flag `-DLS_CO_SCHEDULING_STRATEGY_HARDWARE_CONTENTION` is passed to the compiler's preprocessor when building the library. This makes a process check the shared memory waiting queues of the execution devices that are desired for execution. If any of these queues hold an entry of another process, hardware contention is assumed. The checking process then initiates co-scheduling calculation.

5.1.4 Speedup

Passing the flag `-DDLS_CO_SCHEDULING_STRATEGY_SPEEDUP` to the C preprocessor makes it copy the routine for a optimistic speedup calculation into the check whether a co-

scheduling is necessary. Gaining information about the actual speedup of a co-scheduling requires to calculate the very same thing, which is precisely the procedure desired to be omitted. The optimistic calculation for the speedup therefore replaces a calculation of a co-scheduling. It queries what hardware can be used immediately first, and then compares it with the execution time on the desired hardware devices. If the difference between the desired execution time and the runtime on the free devices is greater than the cost for a co-scheduling, the process of calculating the latter is instantiated.

5.1.5 Waiting Time Amortization

The fourth strategy can be inserted into the code by passing the flag `-DDLS_CO_SCHEDULING_STRATEGY_AMORTIZATION` to the preprocessor when compiling the library. The then inserted feature goes through the waiting queues of all hardware devices that are desired for execution and memorizes the longest waiting time. If the thereby found waiting time is greater than the time it takes to calculate a co-scheduling, the latter is initiated.

```
1  char dls_hwtop_co_scheduling_necessary(struct dls_ll_tasks* tasks){
2
3          //naive approach
4          #ifdef DLS_CO_SCHEDULING_STRATEGY_ALWAYS
5                  return 1;
6          #else
7                  //if there is not at least one task of another process waiting, there is
                       no need for a co-scheduling
8                  if (!dls_hwtop_other_process_has_waiting_tasks())
9                          return 0;
10         #endif
11
12         //co-schedule only if hardware contention is present
13         #ifdef DLS_CO_SCHEDULING_STRATEGY_HARDWARE_CONTENTION
14                 return dls_hwtop_hardware_contention(tasks);
15         #endif
16
17         //co-schedule enables speedup larger than the cost thereof
18         #ifdef DLS_CO_SCHEDULING_STRATEGY_SPEEDUP
19                 return (dls_hwtop_predict_co_scheduling_speedup(tasks) >
                       dls_hwtop_get_co_scheduling_cost());
20         #endif
21
22         //co-schedule if the cost thereof can be amortized by task's waiting time
23         #ifdef DLS_CO_SCHEDULING_STRATEGY_AMORTIZATION
24                 return (dls_hwtop_get_waiting_time(tasks) >
                       dls_hwtop_get_co_scheduling_cost());
25         #else
26                 return 0;
27         #endif
28 }
```

Listing 5.2: Configurable Co-Scheduling Determination

5.2 Inter-Process Communication

This section describes the actual implementation of the necessary inter-process communication. Before the shared memory approach was pursued, a more explicit and also experimental way for processes to exchange information was implemented out, ie. reading another processes' memory region. This turned out meager successfully compared to the implicit communication implementation using data in the shared memory. The following subsections describe issues with the implementation that reads other processes' memory. An overview of possible procedures using the shared memory for information exchange as an implicit communication alternative is given.

5.2.1 Reading other processes' memory

It is possible to read from and write to other processes' memory regions in Linux operating systems, since the contents of their memory are stored in a certain directory: `/proc/<process ID> /mem` and can be opened as a regular file. This can be used to look up data at any address of other processes' logical address space. This subsection points out what problems are faced when choosing to implement this approach for inter-process communication.

Root Privileges

Accessing files in `/proc/` requires root privileges on Linux operating systems. Thus, successfully running a process that tries to access files in this directory requires it to be started with these special rights, making the co-scheduling feature only possible if the user possesses proper privileges. Since this thesis is designed to enable interaction of multiple processes that can be started by any user on a HPC node, all users require these special rights. This might not be in interest of the system administrators, which arises first concern with this approach.

Knowledge of Others Processes' IDs Required

Another problem emerges when it comes to the specific file path where the other process' memory is located at. Since the other processes' IDs need to be known, some inter-process communication needs to precede this approach. A management file in the shared memory which contains the process IDs of all running HALadapt instances can remedy this

circumstance. However, using the shared memory is exactly what is tried to be avoided by this approach.

Knowledge of Others Processes' Data Address Required

The third issue to address is the way how to actually access the other processes' memory. Assuming a process is started with proper privileges and is aware of the process ID whose memory region is to be read, then it is able to obtain a file pointer to the memory of the other process by calling `fopen("/proc/<process ID>/mem, <access mode>)`. Now, within the received file pointer, memory address can be read (using `fread()`) or written (using `fwrite()`), depending on the access mode set when opening the file. Since the desired information are not stored at the beginning of the other processes' memory, it is additionally necessary to know the offset of the desired information. This offset needs to be communicated between processes before making this approach possible. This information can for example also be stored in the shared memory management file mentioned above.

Complexity of Data Structures and Management Overhead

Assuming that the address offset of the desired data in the other process' local memory region is known, one can navigate the file pointer to this address, using `fseek()`. This moves the file pointer to the specified address, allowing to read from or write to data of the other process' memory at a desired address. In order to acquire information for the co-scheduling, data structures like `sturct dls_task` need to be read. This data structure, for example, consists of 26 data fields, 13 of these are pointers to other structures of which 4 are of variable length. All these pointers point to addresses that possibly need be looked up manually using `fseek()` before being able to access their data, that might be data structures again that have pointers again... and so on. This causes a massive management overhead made up by manually navigating through the other process' memory and storing addresses whose content needs to be looked up again. The C language does not have a feature that supports a recursive deep copy of a data structure, that can avoid this cumbersome and error prone writing of code.

Portability

The last concern with this approach is the portability to other operating systems besides Linux. At least Microsoft Windows does not expose the processes' memory in `/proc/<process ID>/mem`. This excludes the co-scheduling feature from other operating systems like Microsoft Windows which results in an unwanted degradation of this thesis.

Conclusion

Comprising the above mentioned facts, manually looking up other processes' memory can not avoid falling back to the shared memory approach, causes an unwieldy management overhead for looking up all the pointers, lists and memory structures, degrades portability and requires root privileges for users. Consequently, this approach has been declared neither suitable nor feasible within the scope of this thesis.

5.2.2 Shared Memory

In contrast to the above presented approach, using the shared memory neither requires users to have root privileges, nor the knowledge of others processes' IDs, nor the knowledge of the address of data of interest in the respective local memory region, nor causes portability issues since there are equivalent functionalities for non-POSIX operating systems[1]. Hence, making use of the shared memory in order to enable inter-process communication is pursued in this thesis.
The shared memory implementation provides another advantage besides the aforementioned ones: it allows incorporation of any external scheduler since all required information is made accessible for all processes in the system. Therefore great extensibility is given. However, this also arises security considerations, as merely deleting the files in the shared memory can cause undefined behavior.
This subsection gives an overview on how shared memory objects are created and managed. Also two different approaches are presented that enable data exchange for co-scheduling tasks of multiple processes.

[1]This feature is also implemented for Microsoft Windows, which is not POSIX compliant.

Usage of POSIX Shared Memory Functions

POSIX compliant shared memory objects are located in `/dev/shm`. Ther are created, or, if already existent, opened by using the POSIX function `shm_open()` which returns a file descriptor for the shared memory object. Its physical address is then mapped into the processes' logical local address space by calling the POSIX function `mmap()`, which returns a pointer to the mapped memory region. This pointer is of type `void*`, ie. the data structure independent type of pointers in C. It can be handled as ("casted to") any type of data. This means that the programmer is not confined in what he intends on doing with the memory pointed to. He is presented a zero-initialized piece of memory to which he can freely store information in any desired data structure.
Passing the flags `S_IROTH` and `S_IWOTHT` to the function `shm_open()`, sets read and write permissions for other processes on creation of the shared memory object. Setting the flags `S_IRUSR` and `S_IWUSR` is also required, since they allow the creating process itself to read from and write to the shared memory object. The combination of these four flags enables read and write permissions for multiple processes to shared memory objects, which implies inter-process communication.

Asynchronous Data Access Handling

Since the shared memory allows multiple processes to have the same physical memory accessible, asynchronous read- or write operations on the same address may occur. This leads in certain situations to data corruption which is desired to be avoided. A very similar issue arises if multiple threads, that, by default, share the same data region, access the same address asynchronously. Therefore the POSIX Threads (`pthread`) execution model provides functions to manage locking mechanisms, so called "Mutexes". These mutexes can also be applied to this process-wide competition for data.

Store All Data in the Shared Memory

HALadapt instances store data structures for submitted tasks in their local memory region. This data structure contains, among data fields, pointers that point to other data structures. Besides one exception, ie. the shared memory waiting queues, all of the data structures pointed to, are also located in the local memory address space. Therefore, allocating a shared memory object for the task data structure alone does not enable other processes to fully access information that would be required to supply their HALadapt built-in task scheduler with all information it needs to find a co-scheduling. Allocating shared memory objects for all the data structures that the task data structure points to leads to the same

problem as pointed out in 5.2.1: need for a recursive deep data copy. Consequently, this approach to supply the built-in task scheduler with information of other processes' tasks is not feasible.

Store Selected Meta-Data in the Shared Memory

With regard to the aforementioned circumstances, this thesis pursues sharing only selected information in the shared memory, ie. exactly that information which is required to co-schedule tasks of multiple processes. Therefore, this thesis enhances the shared memory usage by data structures that are presented in more detail in the following section (5.3).

5.3 Data Structures

According to the above presented reasons, new data structures are introduced that contain the certain payload information that is required for finding a co-scheduling. An additional data structure is invented for management purposes. This section contains details concerning these data structures and explains how they are related to each other and what information they contain. These data structures are located in the shared memory and therefore accessible by any process in the system.

5.3.1 Data Structure “Co-Scheduling Information Management File”

This thesis introduces a management data structure in the shared memory. It is designed to manage the information that is required in order to find a process-wide co-scheduling. It consists of a fixed-length header which contains general information concerning a possibly running or finished co-scheduling calculation. Table 5.1 shows the structure of the CSIMF header.
The so called payload is a variable length list of CSIMF entries which correspond to an CSI files in the shared memory. These files ,along with their use and content, are presented in subsection 5.3.2. Table 5.2 depicts the data structure of the payload, ie. the so called CSIMF entries in the shared memory.

CSIMF Header

The first two data field are dedicated to pthread locking mechanism functions. The "Access Mutex" field is of type pthread_mutex_t and 40 Bytes long. It is the actual mutex that can be acquired by processes. Any other process that intends to access data in the CSIMF needs to acquire this mutex and release it afterwards. This procedure ensures that all asynchronous access to the memory are serialized and no corruption occurs. The second field is 4 Bytes long and of type pthread_mutexattr_t. It can be used to specify further properties of the mutex. Details on this will not be regarded, since they do not matter for the presented implementation.

The third field in CSIMF is the so called "status" field. The status field is used to notify other processes of the current state of a possibly calculated co-scheduling. The status field is set to DLS_CO_SCHEDULING_STATUS_NOT_STARTED, if there is no co-scheduling calculated, or if all processes are done fetching the new schedule for their tasks from the shared memory. The status field is set to DLS_CO_SCHEDULING_STATUS_RUNNING, if a process initiated a co-scheduling and its calculation is still ongoing. Processes that read this state, wait until the calculation of the co-scheduling is completed, which is signaled by DLS_CO_SCHEDULING_STATUS_DONE. This is set, if the process that initiated a co-scheduling is done with the calculation thereof. Processes then fetch the result of the co-scheduling from the shared memory and start execution of the new schedule.

The next field contains the ID of the process that initiated calculating a co-scheduling. Storing this ID is necessary to find a task that is involved in the co-scheduling that is chosen to undertake special action: the "master process" checks if the other processes have fetched the result from the calculated co-scheduling and resets the status field accordingly. This also implies that the other processes have been successfully notified of the new co-scheduling and returned from the execution of their current tasks. The implementation of a detection for that is described in section 5.3.1.

Next in the CSIMF data structure, a variable length list of so called "CSIMF entries" follows. This is also referred to as the "CSIMF Payload".

CSIMF Payload

The CSIMF Payload consists of a variable amount of "CSIMF entries". There is one CSIMF entry for each CSI file that exists in the shared memory currently. This is equal to the sum of processes' tasks that are not in execution currently and therefore subject to

Access Mutex
Mutex Attribute
Status
Master Process' ID
Amount of CSIMF Entries
CSIMF Entry 1
...
CSIMF Entry n

Table 5.1: Header of the CSIMF Data Structure in the Shared Memory.

a co-scheduling.
The CSIMF entry has its identifier in the beginning of its data structure. This identifier consists of the process ID that created the task, a dash ("−")) and the task ID within the process. This identifier corresponds to the file name of the respective CSI entry in the shared memory and is used for other processes to look up the CSI file in `/dev/shm`.
The name field is followed by access mutexes that have similar function as in the CSIMF header. However, in this case, access control does not affect the CSIMF file again, it regulates access to the CSI files. Reason for that is that the CSI file has a header of variable length, that needs to be parsed manually. Reducing data fields within this header means a reduction in error prone manual pointer arithmetic.
The next field in the CSIMF entry is the ID of the process that created this CSIMF entry. This is used to detect whether there are tasks of other processes in the system that can cause need for a co-scheduling. If no presence of other tasks is detected, no co-scheduling is necessary and therefore omitted.
The next field is the file descriptor that has been returned to the process that opened the CSI file. It is used to close the CSI file when the task has been executed. This file descriptor could also be stored in the global local memory of the process. However, based on the idea of separation of concerns [42], global variables are tried to be avoided.
The next two fields are flags, that indicate the status of the task which the CSI file corresponds to. The scheduled flag is set, if the co-scheduler has scheduled the task to an execution device. The ready flag is set if all dependencies have already been scheduled. Both flags are used by the co-scheduler internally. The exact role they play is described in section 5.4.
The next field in the CSIMF entry stores the priority of a task. The priority can be used by the scheduler in order to come to scheduling decisions, eg. which task allocates what accelerator.
The next field contains the round in which the task has been scheduled by the scheduler. It is set by the scheduling process and read by the owning process when fetching the result. It ensures that the processes enqueue their tasks in the correct order to avoid dependency

CSIMF Entry Name
Access Mutex
Mutex Attribute
CSI Owning Process ID
File Descriptor
Scheduled Flag
Ready Flag
Priority
Round
CPU Capability Flag
OMP Capability Flag
GPU Capability Flag
CUDA Capability Flag
OCL Capability Flag

Table 5.2: Data Fields of the CSIMF Entry, also called “CSIMF Payload

violations.
The last fields are flags that indicate to what execution devices a task can be mapped. These flags are used by the co-scheduler to detect tasks that can only run on a specific execution devices and handles them accordingly. For details on their treatment, refer to section 5.4.

5.3.2 Data Structure “Co-Scheduling Information”

Along with the management data structure CSIMF, the actual co-scheduler-relevant data is stored in the shared memory. A data structure called Co-Scheduling Information (CSI) is introduced in this thesis in order to make required information accessible for all processes in the system. The presence of a CSI file implies that a running HALadapt instance has tasks enqueued in the shared memory waiting queues that are not running yet. They are subject to a co-scheduling and are deleted on execution of the respective task. Like the CSIMF, the CSI files consist of a header and a payload. Both of these have variable length. The header contains information on the task’s dependencies and a few other management data fields. The data structure is depicted in table 5.3. The payload contains information on all execution alternatives that this task provides. Its structure is shown in table 5.4.

Number of Dependencies
IDs of Dependencies
Number of Reverse Dependencies
IDs of Reverse Dependencies
“Best” Index
Number of Available Execution Alternatives

Table 5.3: Data Structure the Co-Scheduling Information header

CSI Header

The header of a CSI file starts with the number of dependencies of the respective task. Storing this amount is necessary for correctly parsing the file, since the pointer is moved through it manually. Human readable names of the tasks that this respective task depends on are stored afterwards. They are used by the co-scheduler in order to determine whether a task is ready to be scheduled by looking up all its dependencies and checking their scheduled flag.
Afterwards, the number of reverse dependencies[2] and the names of those are stored. This is necessary because some tasks do have indirect dependencies to other tasks. They need to be resolved, which is done by making use of reverse dependencies. Indirect dependencies may occur if, for example, a memory transfer resides between the execution of two tasks. Then, one of the tasks reversely depends on the memory transfer, whereas the other task depends on it “normally”. They both then store the ID of the memory transfer, which is detected and then replaced by the task ID of the computation task, ie. the direct dependency.
The next data field in the CSI header is called “Best Index”. It is used by the co-scheduler to memorize what entry in the payload is the currently best mapping according to the current co-scheduling state. On termination of the co-scheduling calculation, the respective process looks up this data field in order to fetch the result of the co-scheduling calculation. The last field stores the amount of how many CSI payload entries follow next. This is, once again, necessary for correctly parsing the CSI file.

CSI Payload

The payload of a CSI file can be seen as a table of execution alternatives that this task provides. The entries consist of the respective programming model, eg. OMP, CUDA, etc..., the respective runtime with that programming model and the amount of threads used. The

[2]A task $T0$ is referred to as a reverse dependency of a task $T1$, if $T0$ depends on $T1$.

Programming Model of Implementation 1
Runtime of Implementation 1
Amount of used Threads of Implementation 1
...
Programming Model of Implementation n
Runtime of Implementation n
Amount of used Threads of Implementation n

Table 5.4: Data Structure the Co-Scheduling Information payload

latter is only of interest when the task did not receive an accelerator and was scheduled to the CPU cores.

The general structure of the CSIs is depicted in figure 5.1.

5.4 The Co-Scheduler

Along with the presented shared memory data structures arises the need for a mechanism that can utilize their stored information in order to find a useful co-scheduling. For this, multiple co-schedulers are presented in this thesis. This section focuses on the round based greedy scheduling algorithm, for simplicity referred to as the “co-scheduler”. It fetches its information from the shared memory which allows it to co-schedule tasks of multiple processes. It is accelerator aware and therefore suited for heterogeneous computing system. The primary goal of this scheduler is to make use of all hardware resources that remain idle on a HPC nodes (see 1.1), thus it can be called throughput-driven. It pursues a round based approach for assign tasks to execution devices and tries to allocate 100% of the computing resources that are available in the system. On top of that, it provides configurability for the calculation of task’s priority, that is used to decide which task allocates specific accelerators.

5.4.1 Scheduling Flow

This subsection describes the co-scheduling flow of the presented task co-scheduler. Before the actual co-scheduling begins, a preparation routine is called which acquires more needed information and undertakes modifications on the data in the CSI payload of some tasks. After that, the actual round based scheduling mechanism is repeated until all tasks

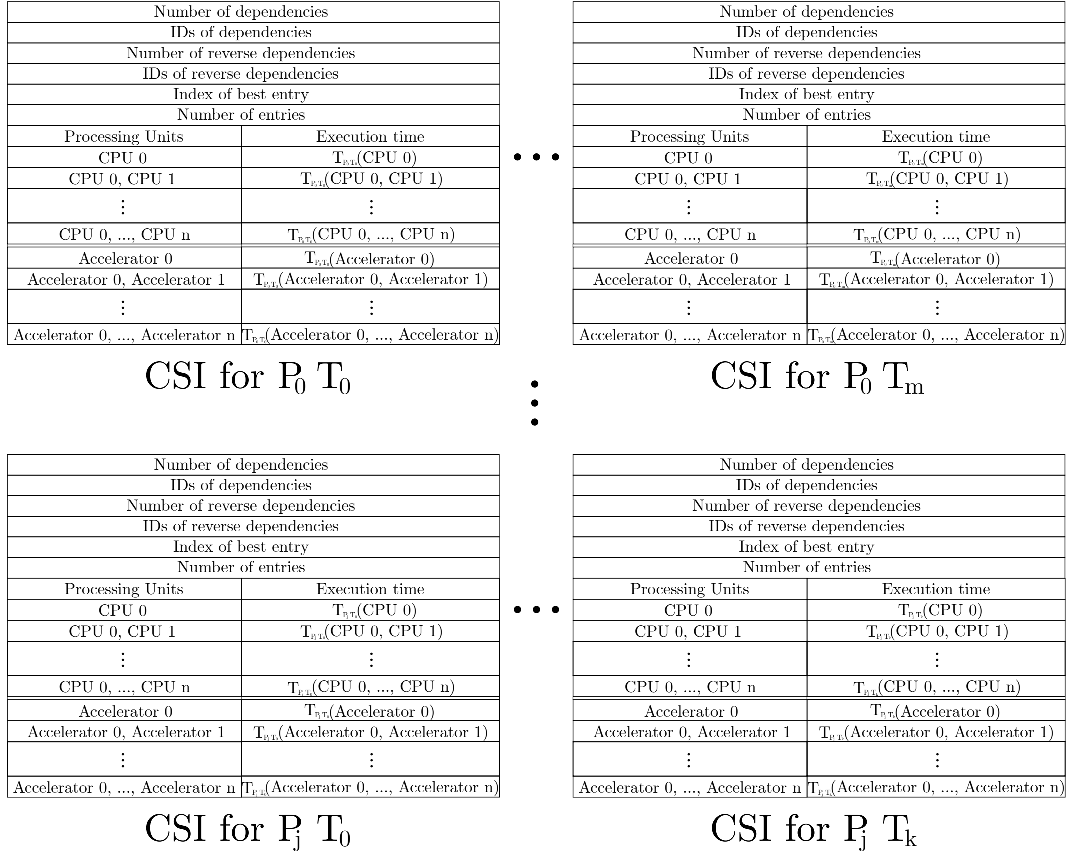

Figure 5.1: CSI files are generated for every task submitted to HALadapt that is queued and not running.

have been assigned to execution devices. The overall procedure is shown in figure 5.2.

Preparation Routine

For obvious reasons, the co-scheduler needs to be aware of the hardware installed in the system. Thus, all available execution devices are counted and grouped by type, eg. CPU, GPU, etc. This allows the scheduler to be aware of tasks that, according to their CSI entries, need to allocate an accelerator or otherwise would be blocked.
Another part of the preparation routine is to resolve indirect dependencies as explained in 5.3.2. In case some tasks have an indirect dependency to another task, the name of the direct dependency is resolved and its name overwrites the name of the indirect dependency. This eases the future handling of dependencies and assures that no non-existing files are tried to be opened in the shared memory.

Round Based Allocation Mechanism

The co-scheduler starts a round based allocation mechanism after the aforementioned preparation routine has been executed. This round based allocation mechanism is applied to all tasks until none is left unscheduled. Each round begins with determining the tasks that are ready for execution. This is done by checking if all their dependencies have been scheduled already. A task can also be detected as ready, if it has no dependencies stored in its CSI header. This task is then assumed to be the first task that a process has submitted and thus is ready to be scheduled. The number of tasks detected to be ready is summed up and used throughout the scheduling round.
As the next step, a feature called “Stickiness” is applied, if exactly one task is ready to be scheduled and other criteria are met. Details on this feature are presented in section 5.4.3. If multiple tasks are ready to be scheduled, or the stickiness criteria of a ready task did not apply, the possible execution devices of the tasks are determined by reading their CSI payload and thereby setting the implementation flags in their CSIMF entries. This enables the co-scheduler to be aware of tasks that solely offer one implementation and therefore need to allocate this particular accelerator. Their execution would be blocked otherwise. A computing device that must be allocated by a task is referred to as an “urgently needed” device. The amount of urgently needed devices is summed up and grouped by type. If this sum for a group of execution device is larger than the amount of respective devices in the system, hardware contention has occurred. In this case at least one task needs to wait. A “winner” needs to be found among all tasks that urgently need this device. This is done by calculating the priority of all competing tasks. The task providing the highest priority

may allocate its urgently needed device. This procedure is repeated until all available accelerators have been allocated. In case the computing system provides enough execution hardware for all demanding tasks, they simply allocate their urgently needed devices.
Any remaining free accelerators in the system are distributed to tasks that can make use of them, ie. offer an implementation for that programming model. If there are more than 1 tasks that could use this accelerator, their priority is consulted to pronounce a "winner" once again. This procedure is repeated until no accelerators that could be used are left unallocated in the system.
All tasks that remain unscheduled at this point, are distributed among CPU threads.

5.4.2 Configurable Priority

Greedy scheduling algorithms that utilize a heuristic for finding a solution have the disadvantage that they can not overcome local extrema. That means, solutions found by such schedulers are possibly not the global optimum, but a local one. Since the presented scheduler is also afflicted by this circumstance, users have the option to configure the priority calculation that is used when accelerators are being distributed among the tasks. This makes the scheduler come to different scheduling solutions that can be more suited for a given problem. Finding a "fair" schedule with heuristic based scheduling algorithms has set up a wide field for research. To name some example fields: parallel computing [43], the Linux kernel [44] [45], multimedia systems which also have real time requirements [46] and many more.

The co-scheduler presented in this thesis supports three different priority definitions:

- **Number of Successors:** The priority is equal to the number of successors of a task. This makes the co-scheduler behave comparable to a list-scheduling algorithm [47] [48].
- **Runtime:** The task with the lowest runtime among all its competing tasks is allowed to allocate an accelerator.
- **Age:** Tasks are accompanied with information about their age. The oldest task that is competing for an accelerator is allowed to allocate the desired device. This can prevent starvation, an issue that scheduling algorithms often have to cope with [49].

The configuration is done in the same way as configuring the determination of a co-scheduling initiation, ie. by passing the respective flag to the preprocessor when building the library. The flags are stored in `local.mk` in the library folder and can be set by adding, or unset by removing, a hashtag ("#") at the beginning of the respective line.

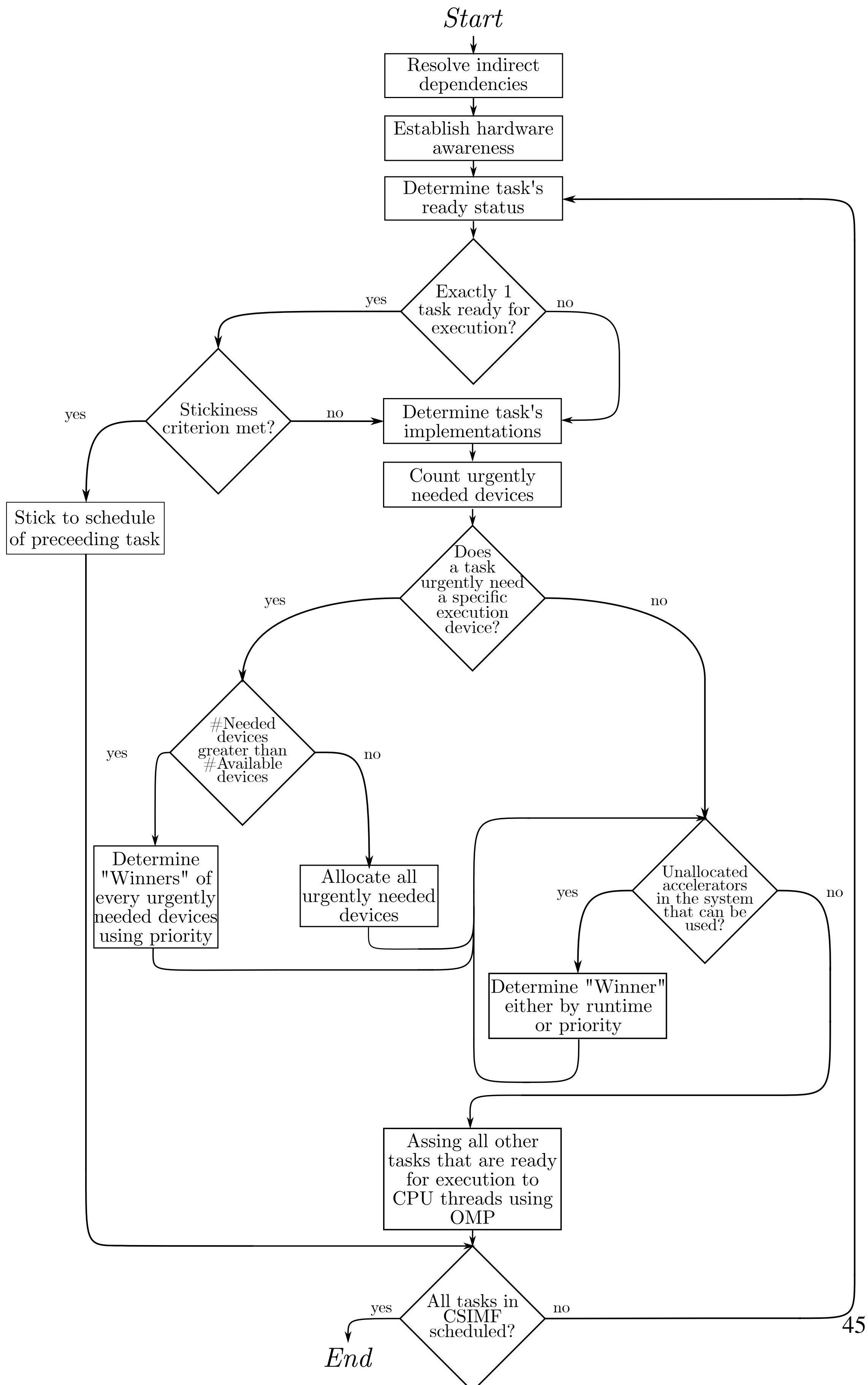

Figure 5.2: Flow diagram of the round based co-scheduler

```
214 ### priority is the number of successors (list scheduling-like behavior)
215 #OPT_DEFS += -DDLS_CO_SCHEDULING_PRIORITY_N_SUCCESSORS
216
217 ### priority determined by the fastest runtime
218 #OPT_DEFS += -DDLS_CO_SCHEDULING_PRIORITY_RUNTIME
219
220 ### use age as priority
221 OPT_DEFS += -DDLS_CO_SCHEDULING_PRIORITY_AGE
```

Listing 5.3: Configuration of the Priority Defenition in File `local.mk`

5.4.3 Stickiness

Another feature that is introduced along with the co-scheduler of this thesis is called "Stickiness". The name originates from its functionality, ie. tasks can stick to an execution device in order to avoid data transfers. Need for this feature was observed while developing the co-scheduler, which mapped single tasks that have been scheduled on the CPU to an accelerator as soon as it became idle. This was often the case for just a single task and caused a memory transfer. Since this behavior is not desired, this feature was developed.

5.5 Random Based Scheduling Algorithms

Besides the round based greedy algorithm, two other scheduling algorithms are introduced. Both of them find their solutions based on random decisions, which is common procedure for generating a starting solution when facing NP-hard problems [41] [50]. In this implementation, the scheduler's solution is not used as a starting point for any succeeding scheduling mechanism. The random based schedulers do only serve for comparison with the round based greedy scheduler and show that the introduced mechanism of information exchange using the shared memory can be used independently from the actual scheduler. They are not likely to run into local extrema, unlike the round based scheduler. The random based schedulers are implemented as an replacement for the evolutionary scheduling mechanisms, which is missing due to time shortage.

5.5.1 Simple Random Mapping Algorithm

The first one is a simple plain random scheduler which merely set the best index of a CSI file to one of its payload entries. Inefficient schedules, idle devices and cache contention

is likely to occur. Since this procedure is not especially promising, an enhanced version of this procedure is presented as well. This scheduling mechanism is referred to as the “enhanced random co-scheduler”.

5.5.2 Enhanced Random Co-Scheduler

The third scheduling mechanism that is presented in this thesis is called “Enhanced Random Co-Scheduler”. It bases, like the mechanism that it is derived from, on random decisions. In contrast, it is equipped with more awareness of on the one hand for available hardware in the system, and on the other hand for the depth of the submitted task graph, ie. the maximum number of successors among all tasks. These pieces of information are taken into account when finding a schedule.

Iterative Procedure

The main idea with this scheduling mechanism is try out multiple random mappings and check if they are “possible” or not. A “possible” mapping is present, if the amount of randomly allocated devices is no larger than the amount of “allowed” devices in the system. The number of “allowed” devices is derived from the depth of the submitted task graph:

$$allowed_devices = available_devices \cdot \max_{t \in tasks} \left\{ \sum_{s \in succ_(t)} 1 \right\}$$

Since this scheduling mechanism is not round based and not aware of any completion time of the tasks, using the maximal amount of successors of all tasks that are to be scheduled is another way to adduce some temporal aspect for the scheduling process.
The enhanced random co-scheduler generates a random task mapping and validates if this mapping is allowed or not. This procedure is repeated multiple times whilst storing the fastest mapping found. The amount of iterations can be configured similar to the other configurable features presented in this thesis.

5.6 Integration in HALadapt

In order to elaborate effectiveness of the presented features, a runtime system that reduces management overhead in developing applications for heterogeneous HPC systems is consulted. This omits cumbersome and error prone tasks like installing and configuring

drivers for execution devices, manual managing thereof, scheduling task executions, writing code that manually manages accelerator usage including memory management and other time consuming issues. In short, programmers can include the HALadapt library in their source code in order to call kernels without taking care for the underlying hardware and mandatory memory transfers that are necessary for a correct execution, significantly reducing programming overhead for heterogeneous computing programs. Therefore, the runtime system HALadapt, which has been developed at KIT by Kicherer et al [25] is consulted. All presented features are integrated into HALadapt and then used to evaluate effectiveness of these features.
This section provides information on the overall procedure of HALadapt, how the features can be en- and disabled as well as a listing of what amendments are necessary to implement correct behavior of the new features.

5.6.1 Activation of Co-Scheduling Features

All presented features are encapsulated by C preprocessor "pragmas" in the source code. This allows the user to configure whether they are desired or not at compile time. The competent flag that needs to be passed to the preprocessor is `DLS_CS` and can be set in the configuration file `local.mk`. This eases to en- or disable the features on demand and is also useful for comparing execution times and measuring caused overhead.

5.6.2 Additionally Required Mechanisms

This subsection describes additional mechanisms that had to be introduced in order to implement a process-wide task co-scheduling.

Inserted Pre-Execution Check

A running HALadapt instance can check if a co-scheduling by another instance is present by reading the status data field of the CSIMF in the shared memory. This check is done once before execution of any task. Since preemption is not designated in this thesis, the flag is not polled while task execution, which could enable an earlier reaction to a present co-scheduling and therefore reduce inefficient hardware usage. However, if a co-scheduling is found while performing this check, the execution of succeeding tasks is skipped. These tasks are remembered as not executed by setting a flag called `rerun` so that further mechanisms are aware of tasks that need to be placed into a new, rebuilt task graph. A global flag called `dls_tgraph_rebuild_necessary` is set as well. This

causes further mechanisms to be engaged after exiting prematurely from this inchoate task graph execution. They are presented in the following.

Looping Mechanism for Task Graph Rebuild

Once a HALadapt instance returns from its task graph execution, the status of the global flag `dls_tgraph_rebuild_necessary` is checked. If it is not set, usual task graph completion is assumed and the HALadapt instance proceeds to exits as usual. However, the procedure is much different if the flag is set at this point, because the need for a task graph rebuild is indicated. A task rebuild routine is executed and a new task execution call is made, closing the looping mechanism. This procedure can be repeated for every arising co-scheduling while task graph execution until it is completely executed.

Code listing 5.4 depicts the implementation of the looping mechanism. The lines preceded by a hashtag (“#”) are preprocessor pragmas which encapsulate the co-scheduling features. First, the HALadapt instance checks if it should initiate a a co-scheduling. The actual loop, implemented as a “do-while” construct, is entered afterwards. Every loop starts by setting 2 flags that manage correct behavior when prematurely returning from task graph execution. Then, the currently existing task graph is executed. This execution might be interrupted by any process’ co-scheduling, even by its own co-scheduling that has been calculated in line 138. As mentioned, returning from task graph execution prematurely causes the flag `dls_tgraph_rebuild_necessary` to be set, which first causes the task graph to be rebuild according to the co-scheduling result in the shared memory, and afterwards causes the loop be run anew in order to execute the new task graph. This procedure is repeated until all tasks in the task graph are executed.

```
136 #ifdef DLS_CS
137         if(dls_cs_co_scheduling_necessary(tgraph -> container -> tasks))
138                 dls_cs_co_scheduling();
139         do {
140                 dls_tgraph_rebuild_necessary = 0;
141                 dls_tgraph_traversed = 0;
142                 dls_task_execute_tgraph(tgraph, 1);
143                 if (dls_tgraph_rebuild_necessary){
144                         dls_cs_rebuild_tgraph(tgraph);
145                 }
146         } while (dls_tgraph_rebuild_necessary);
147 #else
148         dls_task_execute_tgraph(tgraph, 1);
149 #endif
```

Listing 5.4: Looping Mechanism to Enable Any Number of Occurring Co-Schedulings

Task Graph Rebuild

The presence of an instance's co-scheduling invalidates the task graph of all other HALadapt instances. Therefore, an instance needs to be able to rebuild its task graph in case a co-scheduling is detected. For this, a backup of the original task graph is created before its first execution, from which executable copies can be derived. The tasks of the old task graph indicate that they had been skipped, and therefore not executed, by having the flag `rerun` set. All these tasks are inserted into a new task graph, which is then scheduled by one of HALadapt's internal task graph schedulers. They generate mapping candidates among which the best is mapped candidate is chosen for execution and then to the respective hardware devices. Important to note, is that the candidate generation when rebuilding the task graph differs from the one that is run when creating the initial task graph; there is solely one candidate generated, which is fetched from the shared memory. This candidate precisely represents the co-scheduling result. Generating exactly one candidate forces every available scheduler of HALadapt to choose the same mapping, since there are no alternatives generated. This procedure assures a scheduler-invariant way to force the co-scheduling result to take effect in the scheduling of the individual processes' tasks. The thereby created task graph is then executed.

5.6.3 Integration of the Shared Memory Features

HALadapt already makes use of the shared memory and therefore provides functions for managing such objects. These functions are extended to manage the files that are required for co-scheduling multiple processes. For this, on instantiation of a HALadapt instance, the shared memory is checked for presence of already initialized data structures by other HALadapt instances. A routine called `dls_hwtop _open_sm_main()` is run, that opens or creates the needed files as a POSIX shared memory objects in `/dev/shm`. This is done by calling the function `shm_open()`. This function returns a file descriptor for the opened shared memory object which is then passed to `mmap()` to map the shared memory into the local memory space of the process. Another routine called `dls_hwtop_setup_sm_main()` is called afterwards which initializes the data structures, if they have not been filled with information by another HALadapt instance yet. Otherwise, the routine sets pointers to needed information accordingly, making content in the shared memory objects accessible for later use.

5.6.4 Information Needed by the Co-Scheduler

Information stored in the CSI files is extracted from HALadapt's "history", a database which contains information about profiled past executions. Since the Co-Schedulers are designed to read from the CSI files, they can only choose a mapping that has is stored in the history. Therefore, tasks that are subject of a co-scheduling are required to be well profiled by HALadapt. This causes considerable overhead, especially when it comes to varying amount of OMP threads, including executions with a small amount of threads, which are also of interest. They also need to be executed at least once and are likely to have a very high execution time compared to execution times of the same task using an accelerator.
The same need arises when memory transfers are to be taken into account. HALadapt also profiles memory transfers and stores their caused overhead in the database. However, in order to make the Co-Scheduler aware of the transfer time, they need to be profiled at least once, increasing the complexity of a throughout profiling considerably.

5.6.5 Necessary Amendments

This subsection describes necessary amendments that had to be applied to already existing features of HALadapt in order to make it support co-scheduling of multiple processes.

Disabled Loop Detection

HALadapt provides a feature called "Loop Detection", which allows to detect whether the same task is present in the task graph multiple times in a row. If this is the case, a mapping is only calculated for the first task and all its succeeding tasks which are detected to be part of the loop become a reference to the first task. This feature thereby applies changes to the data structure of the submitted tasks, which hinders accessing all tasks individually, which is required when changing their individual mapping in hindsight. In summary, the loop detection confines all tasks within a detected loop to be executed on the exact same mapping as the first task in the loop. Since a co-scheduling can occur while executing the loop, changing the mapping of individual tasks within a loop needs to be possible. Consequently, loop detection is disabled when the co-scheduling feature is desired.

Cleansing of the Waiting Queues

The entries in the shared memory queues serve for multiple purposes. On the one hand, they supply the worker pool with information on what tasks are to be executed, and on the

other hand, are used by other HALadapt instances to be informed about the current state of execution devices. However, if a co-scheduling changes the task mapping, the usage of execution devices changes as well, so not only the worker pool, but also other HALadapt instances should be informed about the new schedule. Therefore, the out of date information in the shared memory waiting queues needs to be replaced by information according to the new schedule. However, HALadapt's worker pool is not designed to abort an execution. This lead to the decision to skip all task executions after the currently running one, if a co-scheduling is detected. The shared memory waiting queues are cleared as a side effect of this design decision.

Memory Retrieval

HALadapt keeps the data required for task execution on the accelerators, unless retrieval is explicitly desired by the user. This has the advantage that data stored there can be reused, if further computations with this data are engaged on the accelerator. The data is transferred automatically to another memory, in case the data is required by a succeeding computation somewhere else. This behavior needs to be amended in order to entirely support the co-scheduling feature, which relies on a completely traversed task graph having all data in the host memory. Therefore, the task graph's exit routine is extended by calling memory transfers for all registered memory regions, that do not reside in the host memory on task graph completion. This ensures that after the task graph is exited prematurely, all data is fetched and ready to be re-distributed to accelerators according to the co-schedule.

6 Evaluation

This chapter presents the evaluation of the implemented features. It serves as an overall assessment of the helping potential for tackling the problem described in 2.5. First, the experimental setup, ie. the hardware on which the evaluation is done, is introduced. The benchmarks used to evaluate the usefulness of the features are described. After that, behavior of the parameterizable enhancements, eg. the co-scheduling determination or the multiple co-scheduling implementations, are evaluated with varyingly chosen parameters in order to establish a differentiated estimation of their usefulness. Lastly, the caused overhead in terms of runtime and memory are evaluated.

6.1 Experimental Setup

This section presents the experimental setup that is used for evaluation.
The HALadapt instances inheriting the presented features are instantiated on server “i82sn05” at KIT’s chair for Computer Architecture and Parallel Processing (CAPP) which is part of the Institute of Computer Science & Engineering (ITEC). The server runs an Ubuntu Linux 18.04 Long Time Support (LTS) version and has CUDA version 10.0.130 and OCL version 2.1 installed.
The server “i82sn05” offers two Intel® Xeon® E5 2650v4 CPUs with 12 cores each running at 2.2 Gigahertz (GHz). The altogether 24 CPU cores offer 48 threads using Intel’s® Hyper Threading (HT) technology. The heterogeneity is introduced by a Nvidia® Tesla® K80, a dual GPU graphics accelerator providing 4,992 CUDA cores. This card can be used by both the CUDA programming model and the OCL programming model.

6.2 Benchmarks

This section introduces the benchmarks that are used in order to generate load in the system. They offer different implementations and different properties in terms of hard-

ware requirement. The Mandelbrot Benchmark is self-implemented, whereas the other two benchmarks, the particle filter and the heat spreading calculation is taken from the Rodinia Benchmark Suite [51].

6.2.1 Mandelbrot Set

A benchmark for data parallel processing is the computation of the Mandelbrot Set. It is computed by iterating equation 6.1 on complex numbers until the absolute value of the complex number is observed to be diverging or converging. These are the termination criteria for the computation. Since all numbers of interest can be iterated on independently, data parallel processors like graphics accelerators enable large speedups compared to an execution using CPUs. For visualization, colors are then derived by mapping the amount of necessary iterations until the iteration was terminated to a color palette, where the color black indicates convergence.
The implementation for this thesis calculates for 72,000,000 complex numbers around the origin of the complex plane, whether they converge or diverge. This procedure makes up one task and can be repeated any number of times.

$$f_c(z) = z^2 + c \qquad (6.1)$$
$$\text{where } z \in \mathbb{C}$$

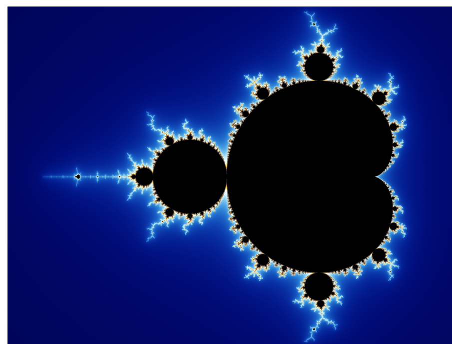

Figure 6.1: The Mandelbrot Set [1]

[1] By Created by Wolfgang Beyer with the program Ultra Fractal 3. - Own work, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=321973

6.2.2 Particle Filter

The source code for the Particle Filter (PF) benchmark is taken from the Rodinia Benchmark Suite and slightly rewritten in order to make use of the HALadapt library for its execution. The PF is a statistical estimator of the location of a target object, that resides within noisy measurement data. The PF has finds application in visual data analysis, ranging from video surveillance in the form of tracking vehicles, cells and faces to video compression.
The PF starts tracking a target object after it has been selected from the noise data environment. For this, guesses about the target's position in the current frame are derived from previous frames' information. A predefined likelihood model is consulted for asses the estimated likelihoods. Those assessments are then normalized and summed up to determine the target object's current location. After that, the guesses are updated, enabling more convincing guesses in the future iterations [52]. This procedure can be, due to Rodinia's input data limitations, repeated up to 10 times. An iteration and consists of 4 tasks each (`likelihood`, `sum`, `normalizeWeights` and `findIndex`), hence this benchmark consists of a maximum of 40 tasks.

6.2.3 Prime Stress

Another benchmark used in this thesis is "Prime Stress", a self implemented prime number finder. The benchmark consists of two parts. First, all prime numbers from 2 to 134,217,728 are determined in parallel using OMP. In the second part, a stress generating calculation is applied to every prime number found. The results of these calculations are then accumulated in a single variable, which is a procedure that can be hardly sped up with parallelism. Consequently, this benchmark inherits a portion that does not scale well with the amount of threads designated for its execution. This property is desired for later evaluation.
One task of this benchmark consists of successively executing both aforementioned parts. The execution can be repeated any number of times.

6.3 Improvements for Computing on Heterogeneous Hardware

This section examines the affect of co-scheduling multiple processes' tasks in heterogeneous computing, ie. involving an accelerator for task execution. Two different aspects

are regarded in a suitable scenario, which is comparable to the one presented in section 2.5.1, in which heterogeneous computing hardware is inefficiently used. Two aspects are examined, ie. the speedup enabled by co-scheduling tasks of multiple processes and secondly, the increased hardware usage that accompanies this procedure.

6.3.1 Scenario Description

This subsection regards the scenario in two processes P_0 and P_1 are started with a small time offset. P_0 consists of multiple tasks that can either be executed on the CPU or on an accelerator, which is preferred since it provides higher throughput than an execution on the CPU threads. The second process, P_1, is started shortly after P_0 has allocated the accelerator. Unlike P_0, P_1 does not offer alternative execution devices. It needs to be executed on the already allocated accelerator. Therefore, hardware contention is faced in this scenario, which can be solved by co-scheduling multiple processes' tasks.
The first process, P_0 is an execution of the Mandelbrot benchmark (see 6.2.1), consisting of 5 tasks. The second process, P_1 is an execution of the PF benchmark (see 6.2.2) which consists of 20 tasks (5 iterations). The time offset is 10 seconds. The script used to timely start the two processes is given in listing 6.1.

6.3.2 Speedup

Figure 6.3.2 shows the execution times of script 6.1 with different features enabled in HALadapt. The dark gray bars represent the execution time of the scripts without any enhancements of this thesis. The green bars show the execution time of the round based scheduler's result. The red and blue bars depict the execution time of the decision of the enhanced random mapping scheduler. There are two different configurations plotted, for 10 iterations and 42 iterations, respectively. Measurements are done for other amounts of iterations (2, 5 and 7) but the execution times of their results are off chart, having execution times up to 10 minutes. Clearly, the more iterations the enhanced random mapping algorithm is allowed to do, the better its results become.
Multiple runs with the same configuration are done in order to cope with runtime fluctuations. The execution of the round based scheduler's decision has an average execution time of 76.0085 seconds, whereas the average execution time of the script without this thesis' features is 96.1865s. This implies a speedup of $96.1865s/76.0085s \approx 126.547\%$. Comparing the most extreme measurements gives an estimation of the largest speedup's magnitude, ie $97.634/71.673 \approx 136.22\%$. Figure 6.3 and 6.4 depict the hardware allocation of the respective processes' tasks.

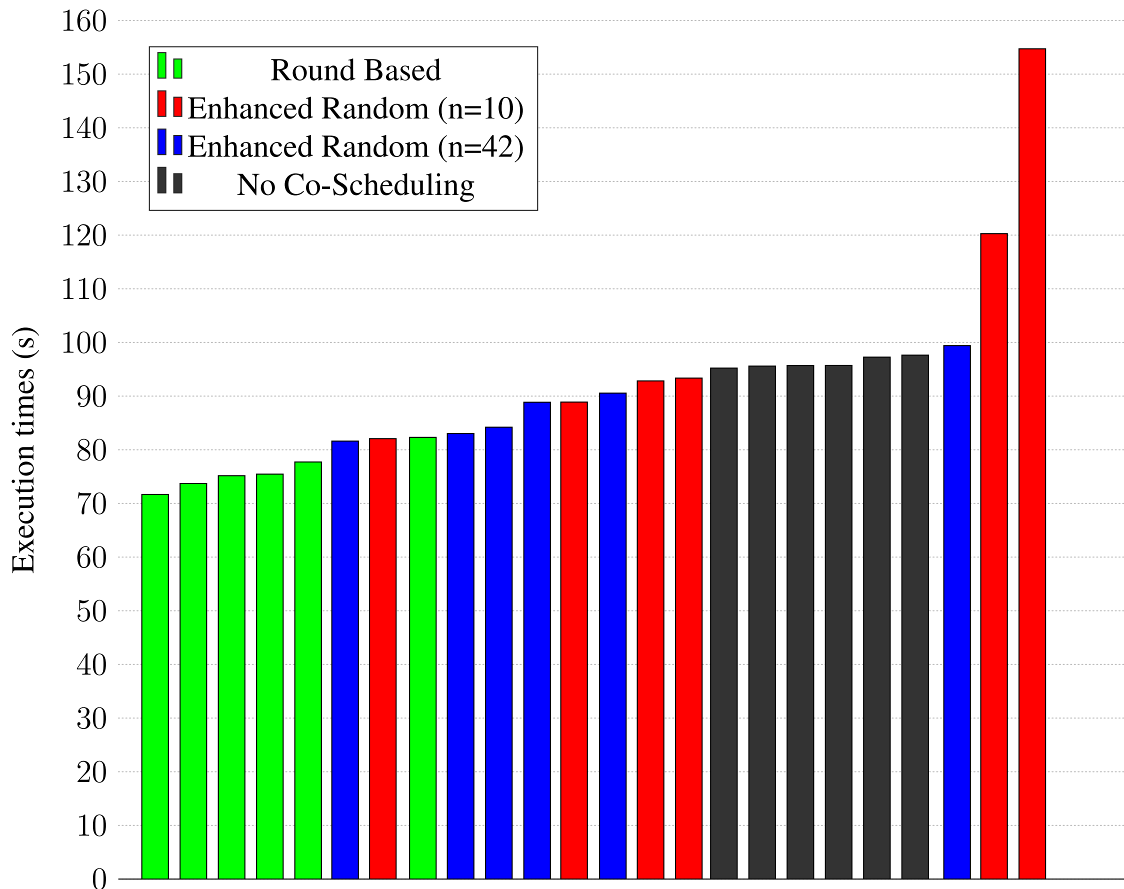

Figure 6.2: Speedup Enabled by Co-Scheduling Multiple Processes

```
1 #start the processes
2 echo "Starting new HALadapt instance (mandelbrot)"; time ../mandelbrot/man && echo "
      HALadapt instance terminated (mandelbrot)" &
3 sleep 10; echo "Starting new HALadapt instance (particle filter)"; time ../particle/
      particle_filter -x 128 -y 128 -z 5 -np 1000000; echo "HALadapt instance terminated (
      particle filter)"
```

Listing 6.1: Script for Evaluating the First Scenario

6.3.3 Increased Hardware Load

Besides the enabled speedup, overall hardware load is increased, too. Only the CUDA device is in use without co-scheduling the tasks, leaving all 48 CPU cores in the system idle. This is equal to 50% hardware load, if the CPU is seen as a monolithic computation device, or otherwise, equal to 2.04%. With the co-scheduling features enabled, the

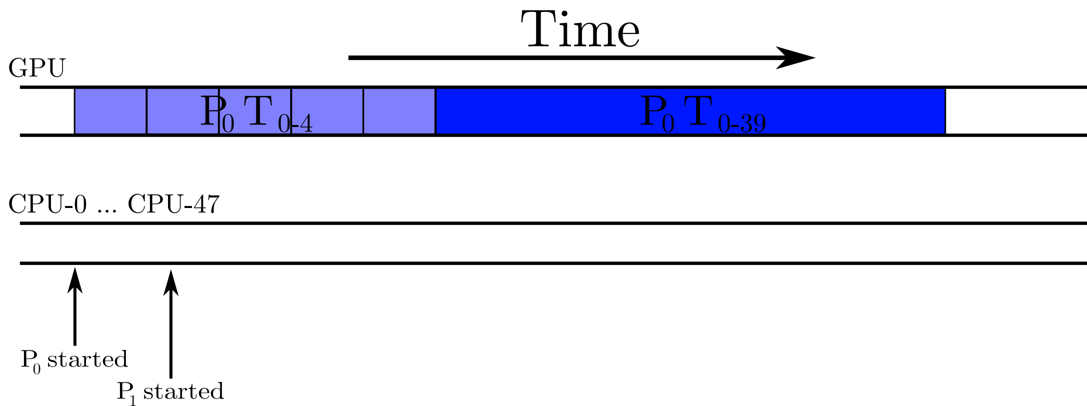

Figure 6.3: Allocation Mapping of Scenario 1 without Co-Scheduling

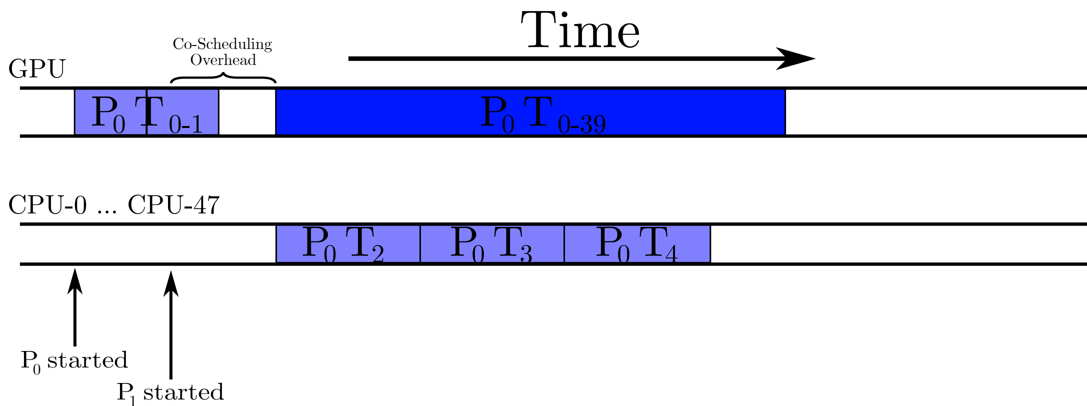

Figure 6.4: Allocation Mapping of Scenario 1 with Co-Scheduling

GPU is in use for averagely 1m9.96s, and then idle until the CPU finishes processing its tasks. This is, in average, 6.0432s later the case, implying an overall hardware load of $1 \cdot (69.96s/76.0085s) + 0.5 \cdot ((76.0085s - 69.96s)/76.0085s) \approx 96.01\%$, or if the CPU is not regarded as a monolithic device: 99.83%.

6.4 Improvements for Computing on Homogeneous Hardware

Co-scheduling multiple processes' tasks also can be applied in the absence of accelerators in the system. This section examines the effect of co-scheduling tasks that can only

run on the CPU. As shown in section 2.3, computational problems do not benefit from arbitrarily large amounts of parallelism. Therefore, it can be useful to take task's scaling properties into account and distribute CPU cores accordingly. This procedure can also prevent tasks from being blocked, in case no CPU cores are available and if there is no way to re-arrange their distribution.

6.4.1 Scenario Description

The following case is set up for this section: a process P_0 is started and submits its tasks into the runtime system. These tasks can only make use of the CPU, which are all idle when P_0 is started. In order to provide highest throughput, up to all CPU cores are allocated for executing P_0's tasks. However, the tasks do not scale very well with the high amount of allocated threads. Another process, P_1 is stared shortly after the execution of P_0's tasks is engaged. P_1's tasks have the same characteristics as P_0's tasks and either have to wait until P_0's tasks are finished or use idle CPU cores, if any.

The benchmark used for evaluating this scenario is "Prime Stress" (see 6.2.3) since it inherits a considerably portion of code that is not scaling with parallelism. The benchmark is repeated four times, making up four tasks in total per process.
The round based greedy co-scheduler is used in this evaluation. The co-scheduling criterion is set to "Hardware Contention".

6.4.2 Increased Fairness

Listing 6.2 shows the script used for evaluating this scenario. Two processes are started, both executing the Prime Stress benchmark, with a time offset of 33 seconds. Measuring the execution time of the script does, in contrast to the heterogeneous evaluation case, not show a speedup. In fact, the sum of the processes' execution time is always greater when initiating a co-schedule. However, another interesting aspect arises when regarding the processes' runtime: since the round based greedy scheduler distributes CPU threads evenly to all tasks that did not allocate an accelerator, the difference (Δ) of the two processes' execution time is lowered significantly. This implies an increase in fairness, ie. all processes face the same slowdown due to allocated execution devices by other processes[2].
Figure 6.6 and 6.7 depict the hardware allocation of the respective processes' tasks.

[2]Note that the evaluation node was in use by others during conducting these measurements. Possible measuring distortions are tried to be amortized by averaging results of multiple runs.

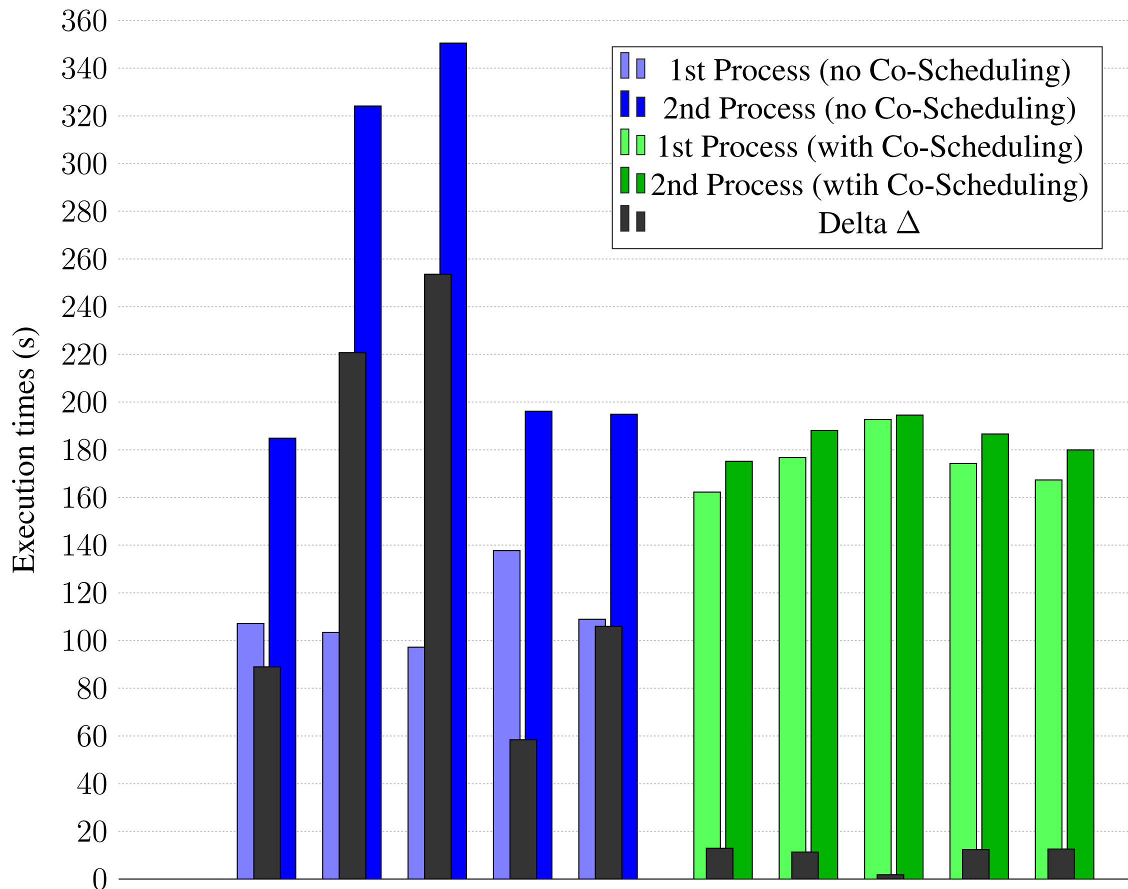

Figure 6.5: Improved Fairness by Co-Scheduling Multiple Processes

```
1  #start the processes
2  echo "Starting new HALadapt instance (primestrss0)"; time echo "r" | gdb ../prim/prim &&
       echo "HALadapt instance terminated (primestress0)" &
3  sleep 33; echo "Starting new HALadapt instance (primestress1)"; time echo "r" | gdb ../
       prim/prim; echo "HALadapt instance terminated (primestress1)"
```

Listing 6.2: Script for Evaluating the Second Scenario

6.5 Runtime Overhead

This section sheds light upon the runtime overhead caused by executing the presented features. First, the four co-scheduling determination strategies are evaluated, followed by the different co-scheduling procedures.
HALadapt provides built-in profiling mechanisms that can be used to measure time passed

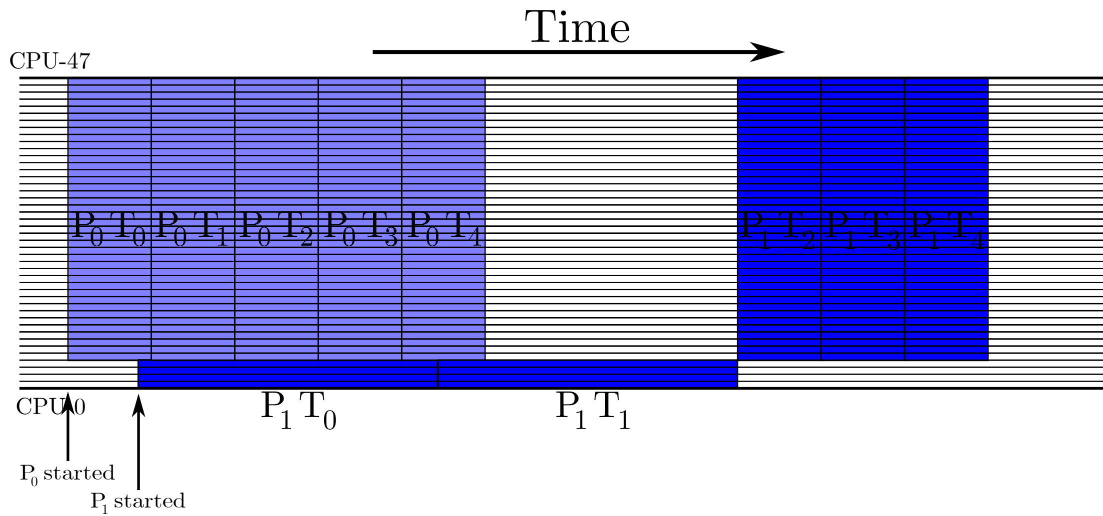

Figure 6.6: Allocation Mapping of Scenario 2 without Co-Scheduling

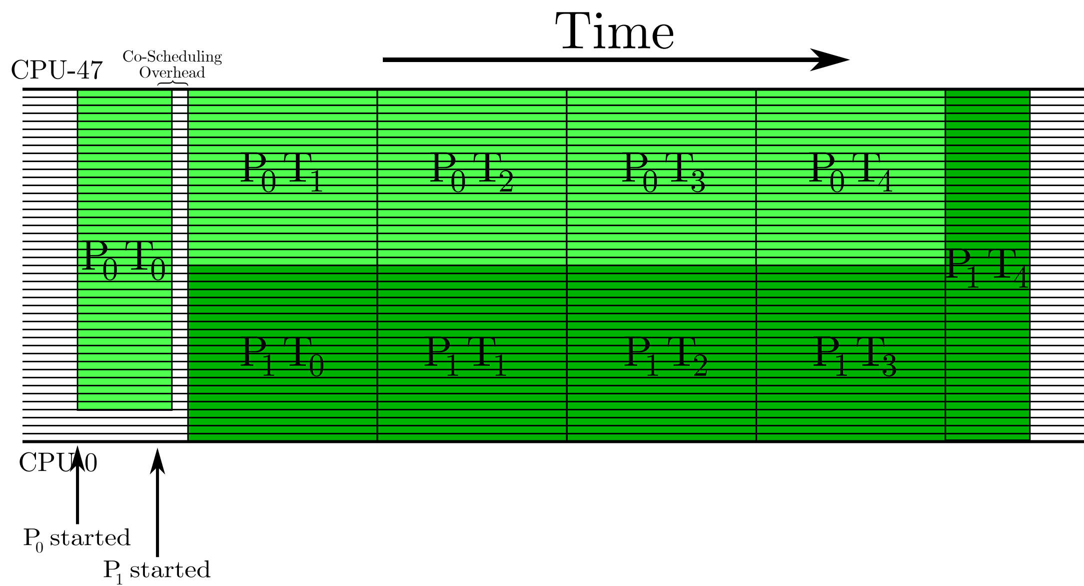

Figure 6.7: Allocation Mapping of Scenario 2 with Co-Scheduling

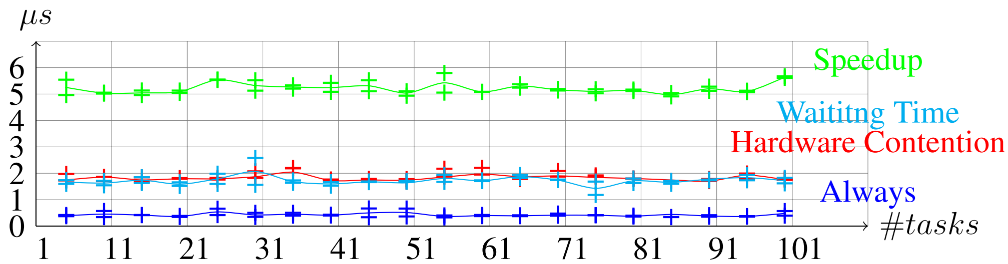

Figure 6.8: Runtime Overhead of Co-Scheduling Determination Strategies

between lines of code, allowing for precise measurement of function execution. The measurements are committed multiple times in order to cope with hardly predicable fluctuation that occurs when measuring runtime. The thereby resulting measurements are then averaged. Regression is then applied to the averaged values in the later sections, so that a general growth can approximately be derived from the measured runtimes.

6.5.1 Runtime Overhead of the Co-Scheduling Determination Strategies

This subsection examines the runtime overhead caused by the respective co-scheduling strategies. Measurements are done for varying amount of tasks in the system in order to establish awareness of the runtime overhead caused by the amount of tasks. This is useful, since some of the co-scheduling determination strategies require information from these tasks. Therefore, the runtime overhead can vary with the amount of tasks in the system.

6.5.2 Runtime Overhead of the Co-Schedulers

This subsection presents the runtime overhead that is caused by the introduced scheduling mechanisms. The round based greedy scheduler is evaluated first. All of its configurable priority definitions are evaluated individually. After that, the two random based schedulers are evaluated.
The measurements are committed multiple times in order to cope with hardly predicable fluctuation that occurs when measuring runtime. The measurements are averaged before quadratic regression is applied, which provides information about the overall growth of runtime overhead caused by a single task. Quadratic regression is chosen, since the scheduler's priority calculation procedure inherits quadratic runtime complexity. This is due to

Priority Definition	Quadratic Regression	Correlation Coefficient r
Age	$398.4 \cdot x^2 + 61,230 \cdot x + 232,973$	0.999767
Runtime	$379.5 \cdot x^2 + 126,480 \cdot x + 152,069$	0.999324
Number of Successors	$398.2 \cdot x^2 + 71,221 \cdot x + 174,260$	0.999737

Table 6.1: Quadratic Regression Results for the Priority Definitions' Runtime Overhead

comparing on task with all other ones pairwise, which implies $O(n^2)$ comparisons.

Round Based Greedy Scheduler

At first, the runtime caused by the round based greedy scheduler and its three priority definitions, ie. runtime, age and number of successors is measured in order to establish awareness of their runtime overhead. These measurements are committed without the stickiness feature enabled, since it would only take affect if the amount of tasks to co-schedule is equal to 1.
Table 6.1 depicts the results of the quadratic regression for the averaged measurements of the respective priority definition. Figure 6.9 plots the corresponding quadratic regression curves and the measured runtimes which are used to calculated the averaged values for the regression.

Random Based Schedulers

The two presented random based schedulers are evaluated in the following. The plain random mapping scheduler does not offer any configurability, unlike the enhanced random mapping scheduler and the round based greedy scheduler. The enhanced random based mapping algorithm offers configurability in terms of the minimum amount of iterations for finding a valid solution. This feature is evaluated with different minimum amount of iterations. Since the actual amount of iterations is random, any regression is hardly useful for concluding a overall growth in runtime overhead. However, since the scheduling algorithm shows, in spite of the random amount of iterations, a quadratic runtime scaling, regression is applied and plotted in 6.10 alongside with the raw measurements for a minimum amount of 2, 10 and 30 iterations.
Quadratic regression is also applied to the averaged measurements of the plain random mapping algorithm and plotted in figure 6.11. The quadratic regression's results are presented in table 6.2.

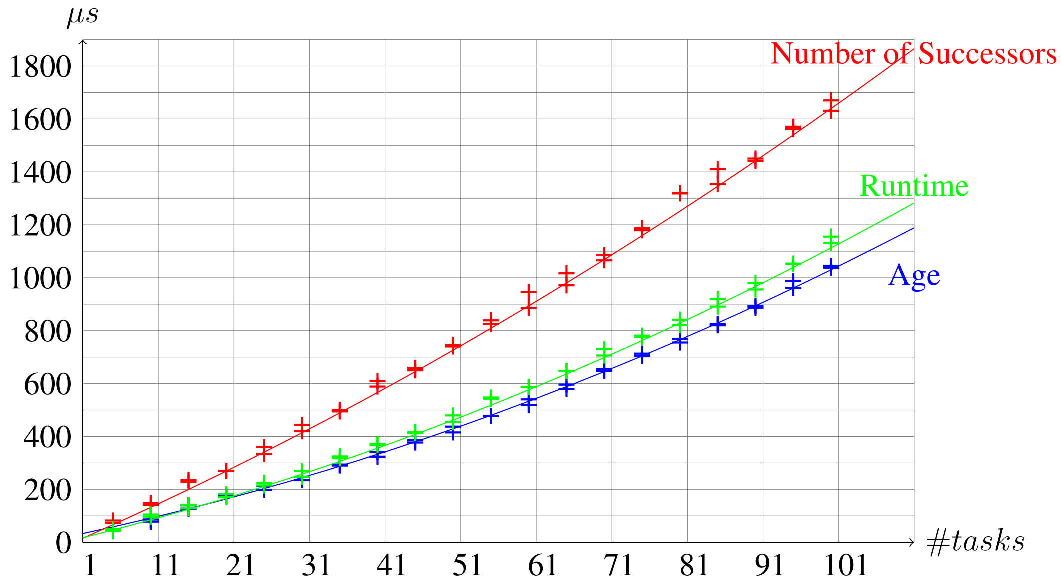

Figure 6.9: Runtime Overhead of the Round Based Scheduler's Priority Definitions

Random Mapping Algorithm	Quadratic Regression	C.C.[3] r
Enhanced (min. it. = 2)	$6,394 \cdot x^2 - 70,882 \cdot x + 2,753,900$	0.986696
Enhanced (min. it. = 10)	$8,491 \cdot x^2 + 248.572 \cdot x + 2,072,000$	0.971705
Enhanced (min. it. = 30)	$10,911 \cdot x^2 + 504,508 \cdot x + 5,254,200$	0.985732
Plain	$330.7 \cdot x^2 + 13,240 \cdot x - 2,814$	0.99942

Table 6.2: Quadratic Regression Results for the Plain Random Mapping Runtime Overhead

6.6 Memory Overhead

This section examines the memory overhead caused by enabling co-scheduling capabilities. Since the introduced memory structures need space to reside in, the extent of higher memory utilization is to be examined. This section will analyze the memory overhead caused by both data structures that are introduced, ie. the CSIMF and the CSI files.

[3] Correlation Coefficient

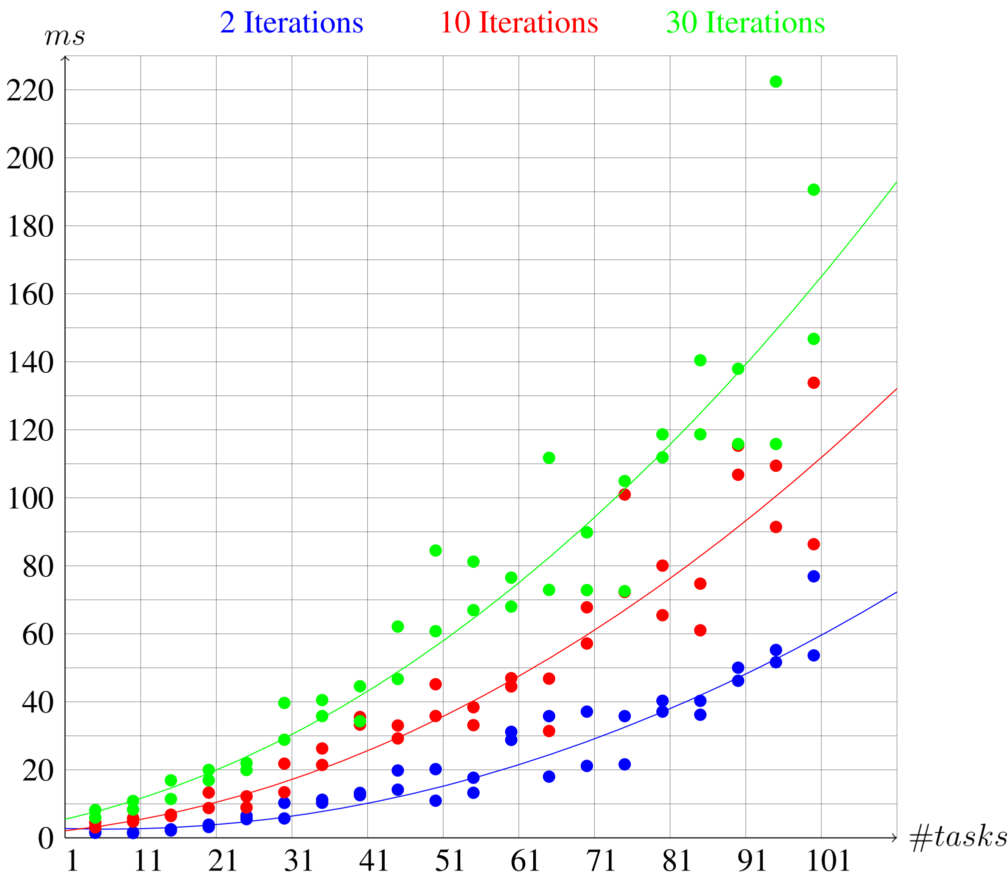

Figure 6.10: Runtime Overhead of the Enhanced Random Based Scheduler

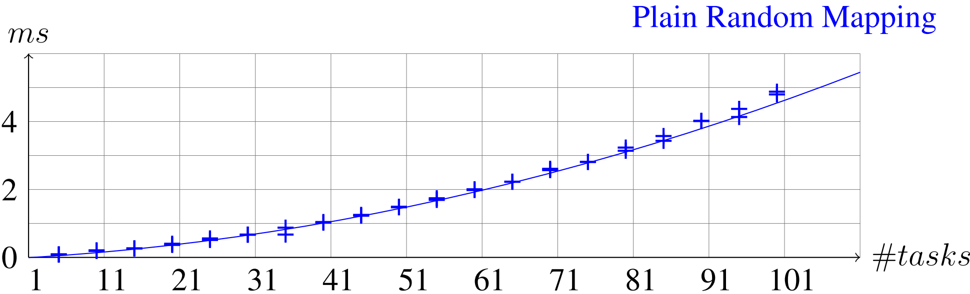

Figure 6.11: Runtime Overhead of the Plain Random Based Scheduler

Name	Type	Size (Bytes)
Access Mutex	`pthread_mutex_t`	40B
Mutex Attribute	`pthread_mutexattr_t`	4B
Status	`int`	4B
Master Process' ID	`int`	4B
Amount of CSIMF Entries	`size_t`	8B
Sum		64B

Table 6.3: Memory Requirement of CSIMF's Header

6.6.1 Memory Overhead Caused by the Co-Scheduling Information Management File

The memory requirement for the CSIMF is the sum of all its header fields and the sum of the fields in the payload times the amount of entries that are currently stored in the CSIMF. The amount of CSIMF entries is equal to the number of tasks that all HALadapt instances in the system have enqueue and not running. The memory requirement for the header is independent from the state of the system and has a fixed length. The sizes of the header's data fields are given in table 6.3. The length of CSIMF's payload is fixed, too. However, its amount varies with the state of the system. The memory requirement is depicted in table 6.4.
The maximum size of the CSIMF has been statically set to 256KiB, which leaves 262,080 Bytes for CSIMF entries. This allows up to 1,885 CSIMF entries to fit into the CSIMF. Consequently, 1,885 tasks can reside alongside in the CSIMF and can theoretically be co-scheduled. The size can be in- or decreased at compile time, allowing for a lighter memory requirement, or space for more co-existing tasks.

6.6.2 Memory Overhead Caused by the Co-Scheduling Information

The memory requirement for the CSI files can be calculated by summing the size of its data fields up. Important is the variable length of both the CSI file's header and the varying amount of payload entries. The header varies in length since it depends on the amount of dependencies and reverse dependencies of the respective task, whereas the payload varies with the amount of entries in HALadapt's history database for the respective task. The header fields and their size is given table 6.5. Note that the stated sum of 32 bytes is equal to the minimum size of a CSI file's header, namely the one without any dependencies or reverse dependencies. However, an average of 3 dependencies and 1 reverse dependency

Name	Type	Size (Bytes)
Name	`char[DLS_NAME_LENGTH]`	64B
Access Mutex	`pthread_mutex_t`	40B
Mutex Attribute	`pthread_mutexattr_t`	4B
Owner's Process ID	`int`	4B
File Descriptor	`int`	4B
Scheduled Flag	`char`	1B
Ready Flag	`char`	1B
Priority	`size_t`	8B
Round	`size_t`	8B
Time of Creation	`dls_time_t`	8B
Device Offset	`unsigned int`	4B
Has CPU Flag	`char`	1B
Has OMP Flag	`char`	1B
Has GPU Flag	`char`	1B
Has CUDA Flag	`char`	1B
Has OCL Flag	`char`	1B
Sum		139B

Table 6.4: Memory Requirement of CSIMF's Payload

is observed when using HALadapt's internal task graph scheduler. Therefore, a more realistic memory requirement of $8B + (3 \cdot 64B) + 8B + (1 \cdot 64B) + 8B + 8B = 288B$ is observed.
In contrast, the payload of a CSI file has a fixed length. Nevertheless, the amount of stored entries varies. In practice, no more entries than 50 are observed. However, this depends on the amount of execution devices in the system as well as on the extent of profiled executions. Therefore, the size of a CSI has been chosen to be 2KiB, offering 560 bytes more space than required in the aforementioned scenario. Likewise the CSIMF, the CSI file's size can be adjusted at compile time in order to fit special requirements. The CSI's payload size if given in 6.6.

6.6.3 Overall Memory Overhead

Comprising the above shown facts, the memory requirement of the co-scheduling features is at least the sum of the size of the mandatory CSIMF and one CSI file, since there can be no co-scheduling without any task. Therefore, a minimal overall memory requirement is equal to $256KiB + 2KiB = 264,192B$. As the interesting cases contain more than one

Name	Type	Size (Bytes)
Amount of Dependencies	`size_t`	8B
ID of Dependency 1...n	`char[DLS_NAME_LENGTH]`	$\{64B\}^*$
Amount of Reverse Dependencies	`size_t`	8B
ID of Reverse Dependency 1...m	`char[DLS_NAME_LENGTH]`	$\{64B\}^*$
Best Index	`size_t`	8B
Amount of Payload Entries	`size_t`	8B
Sum		≥32B

Table 6.5: Memory Requirement of CSI's Header

Name	Type	Size (Bytes)
Programming Model	`enum impl_pmodel`	4B
Runtime	`dls_hist_key`	8B
Threads	`dls_hist_key`	8B
Sum		20B

Table 6.6: Memory Requirement of CSI's Payload

task in the system, the memory requirement is greater than that number in practice. It can be described by following equation: $mem_overhead = 256KiB + (2KiB \cdot \#tasks)$

7 Summary

This chapter serves as a recapitulation of the topic and purpose of this thesis. The problem to address is explained briefly before explaining the methodology used in order to reach the thesis' goals. Lastly, an outlook on what can be further done, based upon the hereby work presented.

7.1 Co-Scheduling of Multiple Processes in Heterogeneous Systems

Increasing demand for computational power whilst facing physical limitations has led to use of massive on-node parallelism in high performance computing. So called heterogeneous nodes are equipped with parallel architectures, or accelerators, in order to increase system's throughput. However, since not all computational tasks can benefit from the high degree of parallelism, inefficient hardware usage is likely to occur. This thesis proposes making use of multiple processes' inherent data independence from each other to increase computing node utilization by co-scheduling them to the same computation node. Approach and implementation of mechanisms for establishing inter-process task awareness, along with a co-scheduler that enables process-wide re-arrangement of these tasks is presented. Measurements show possible speedups up to 136.22% while making nearly full use of all hardware resources in case of heterogeneous computing, and increased hardware usage as well as a more fair hardware allocation in the homogeneous case.

7.2 Goals Reached

This section recapitulates the goals from section 1.2 and presents what has been done for their achievement. The goals are:

1. Detect inefficient schedule of multiple processes' tasks
2. Determine point in time when to co-schedule, preventing large overhead

3. Enable inter-process communication
4. Create data structures that contain data required for a useful co-scheduling
5. Introduce mechanism that finds a co-schedule using the customized data structures
6. Implement these mechanisms into the runtime system “HALadapt”
7. Demonstrate enabled speed up by running suitable benchmarks

Goals 1 and 2 have been combined by introducing 4 different mechanisms that allow detection of a possible better co-scheduling of multiple processes’ tasks and, based on their policy, also the determination for need of a co-scheduling.
Inter-process communication is achieved by making use of the shared memory. This also allows for any other application to access the data used for a co-scheduling, allowing for example external scheduling applications to compute a co-schedule.
Required data structures and their management are introduced by this thesis. The reside in the system’s shared memory, as mentioned before.
Multiple co-schedulers are implemented, comprising a greedy algorithm and two random based mapping procedures. They use the introduced shared memory data structures for receiving data from the processes and also for instructing the processes to adhere to the new schedule. This task could also be offloaded to any scheduling application that handles the introduced shared memory data structures properly.
For evaluation purposes, the co-scheduling features has been integrated into the runtime system “HALadapt”. Some amendments to it were necessary to support the features as desired.
Benchmarks show a possible speedup of 136.22% when co-scheduling processes’ tasks compared to a conventional execution. The system’s hardware load is thereby increased significantly.

In retrospection, all goals set for this thesis are reached. On top, faster executions are enabled and an extensible interface for co-scheduling tasks of multiple processes in heterogeneous computing is achieved.

7.3 Outlook

This section briefly outlines on further effort can be spent. These are fields that come up alongside with the presented, implemented and evaluated ideas.

Security Enhancements

Since the shared memory is used for inter-process communication, every process has access to the co-scheduling information. This is, on the one hand, a desired property, but on the other hand arising security issues. Deleting the CSIMF or any CSI file causes undefined behavior of the HALadapt instances. Unless users are sufficiently trusted and instructed to proper usage of the shared memory files, security mechanisms should be considered. Marking the shared memory files as read-only also restricts the HALadapt instances from creating their CSI files and registering them in the CSIMF. Selective file access for user groups is an option, but introduces nonadministrative overhead.

Adding Schedulers

The shared memory interface is scheduler invariant. Any scheduling mechanism can read from the CSIMF and CSI files and write its schedule to them. This thesis does not focus on finding an optimal schedule, and the round based scheduler is likely to run into local extrema. A evolutionary scheduling algorithm was desired to be implemented in this thesis, but was omitted due to time shortage. Its implementation is pending.

Optimizing Task Graph Exit

The premature task graph exit does not omit memory allocations on the accelerator that has been chosen for execution in the outdated task graph. Merely skipping them does not notify tasks that depend on this memory allocation, ending up with a blocked task graph exit. More effort can be spent on optimizing the premature task graph exit procedure. However, this requires deeper knowledge of the internal locking an signaling mechanisms of HALadapt.

More Detailed Evaluation

Evaluation of the co-scheduling determination criteria has not been realized thoroughly. Finding a suitable scenario in which the respective determination criteria shows different, measurable and reproducible behavior is pending. At this point, one can only derive from the evaluation that the runtime overhead of the respective co-scheduling criteria is vanishingly low.

The evaluation of the stickiness feature and the co-scheduling impact with more than two processes is pending also.

Appendix

List of Abbreviations

AMD Advanced Micro Devices

API Application Programming Interface

ASIC Application Specific Integrated Circiut

CAPP Computer Architecture and Parallel Processing

CMOS Complementary Metal-Oxide-Semiconductor

CMP Chip Multiprocessor

CPU Central Processing Unit

CSI Co-Scheduling Information

CSIMF Co-Scheduling Information Management File

CUDA Computing Unified Device Architecture

DAG Directed Acyclic Graph

FIFO First In, First Out

FPGA Field Programmable Gate Array

GHz Gigahertz

GPU Graphics Processing Unit

HEFT Heterogeneous Earliest Finish Time

HPC High Performance Computing

HT Hyper Threading

I/O Input / Output

ILP Instruction Level Parallelism

IPC Instructions per Cycle

IPCF Inter-Process Communication File

IPM Integer Programming Model

ITEC Institute of Computer Science & Engineering

KIT Karlsruhe Institute of Technology

LAMA Library for Accelerated Math Applications

LTS Long Time Support

LWP Light-Weight Process

MAP Multidimensional Assignment Problem

MLEM Maximum Likelihood Expectation Maximization

MPI Message Passing Interface

OCL Open Computing Language

OMP Open Multi-Processing

OpenSURF Open source Speeded Up Robust Feature

PF Particle Filter

POSIX Portable Operating System Interface

Bibliography

[1] BUCHTY, Rainer ; HEUVELINE, Vincent ; KARL, Wolfgang ; WEISS, Jan-Philipp: A Survey on Hardware-aware and Heterogeneous Computing on Multicore Processors and Accelerators. In: *Concurr. Comput. : Pract. Exper.* 24 (2012), Mai, Nr. 7, 663–675. http://dx.doi.org/10.1002/cpe.1904. – DOI 10.1002/cpe.1904. – ISSN 1532–0626

[2] DHRUBA CHANDRA ; FEI GUO ; SEONGBEOM KIM ; YAN SOLIHIN: Predicting inter-thread cache contention on a chip multi-processor architecture. In: *11th International Symposium on High-Performance Computer Architecture*, 2005. – ISSN 1530–0897, S. 340–351

[3] A. FEDOROVA ; M. SELTZER ; C. SMALL ; D. NUSSBAUM: Performance of multithreaded chip multiprocessors and implications for operating system design. In: *n Proceedings of USENIX Annual Technical Conference*, 2005, S. 395–398

[4] CHANDRA, Dhruba ; GUO, Fei ; KIM, Seongbeom ; SOLIHIN, Yan: Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. In: *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. Washington, DC, USA : IEEE Computer Society, 2005 (HPCA '05). – ISBN 0–7695–2275–0, 340–351

[5] MUNSHI, A.: The OpenCL specification. In: *2009 IEEE Hot Chips 21 Symposium (HCS)*, 2009, S. 1–314

[6] The lost history of the transistor. In: *IEEE Spectrum* 41 (2004), May, Nr. 5, S. 44–49. http://dx.doi.org/10.1109/MSPEC.2004.1296014. – DOI 10.1109/M-SPEC.2004.1296014. – ISSN 0018–9235

[7] KILBY, J. S.: Invention of the integrated circuit. In: *IEEE Transactions on Electron Devices* 23 (1976), July, Nr. 7, S. 648–654. http://dx.doi.org/10.1109/T-ED.1976.18467. – DOI 10.1109/T–ED.1976.18467. – ISSN 0018–9383

[8] *Where Can I Find Information about the Founders of Intel?* https://www.intel.com/content/www/us/en/support/articles/000015012/programs.html, . – Accessed: 2019-05-08

[9] MOORE, G. E.: Cramming More Components Onto Integrated Circuits. In: *Proceedings of the IEEE* 86 (1998), Jan, Nr. 1, S. 82–85. http://dx.doi.org/10.1109/JPROC.1998.658762. – DOI 10.1109/JPROC.1998.658762. – ISSN 0018–9219

[10] MOORE, G. E.: Progress in digital integrated electronics [Technical literaiture, Copyright 1975 IEEE. Reprinted, with permission. Technical Digest. International Electron Devices Meeting, IEEE, 1975, pp. 11-13.]. In: *IEEE Solid-State Circuits Society Newsletter* 11 (2006), Sep., Nr. 3, S. 36–37. http://dx.doi.org/10.1109/N-SSC.2006.4804410. – DOI 10.1109/N–SSC.2006.4804410. – ISSN 1098–4232

[11] STEEHLER, Jack K.: Understanding Moore's Law–Four Decades of Innovation (David C. Brock, ed.). In: *Journal of Chemical Education* 84 (2007), Aug, Nr. 8, 1278. http://dx.doi.org/10.1021/ed084p1278. – DOI 10.1021/ed084p1278. – ISSN 0021–9584

[12] DUBASH, Manek: *Moore's Law is dead, says Gordon Moore*. https://www.techworld.com/news/tech-innovation/moores-law-is-dead-says-gordon-moore-3576581, 4 2010. – Accessed: 2019-05-15

[13] COLINGE, Jean-Pierre: Nanowire Quantum Effects in Trigate SOI MOSFETs. In: HALL, Steve (Hrsg.) ; NAZAROV, Alexei N. (Hrsg.) ; LYSENKO, Vladimir S. (Hrsg.): *Nanoscaled Semiconductor-on-Insulator Structures and Devices*. Dordrecht : Springer Netherlands, 2007. – ISBN 978–1–4020–6380–0, S. 129–142

[14] BURTON, Graeme: *TSMC says 3nm plant could cost it more than $20bn*. https://www.theinquirer.net/inquirer/news/3018890/tsmc-says-3nm-plant-could-cost-it-more-than-usd20bn, 10 2017. – Accessed: 2019-05-15

[15] DENNARD, R. H. ; GAENSSLEN, F. H. ; RIDEOUT, V. L. ; BASSOUS, E. ; LEBLANC, A. R.: Design of ion-implanted MOSFET's with very small physical dimensions. In: *IEEE Journal of Solid-State Circuits* 9 (1974), Oct, Nr. 5, S. 256–268. http://dx.doi.org/10.1109/JSSC.1974.1050511. – DOI 10.1109/JSSC.1974.1050511. – ISSN 0018–9200

[16] RAGHUNATHAN, Anand ; JHA, Niraj K. ; DEY, Sujit: *High-Level Power Analysis and Optimization*. Kluwer, 1998 http://www.springer.com/engineering/circuits+%26+systems/book/978-0-7923-8073-3?changeHeader. – ISBN 978–0–7923–8073–3

[17] KIM, N. S. ; AUSTIN, T. ; BAAUW, D. ; MUDGE, T. ; FLAUTNER, K. ; HU, J. S. ; IRWIN, M. J. ; KANDEMIR, M. ; NARAYANAN, V.: Leakage current: Moore's law meets static power. In: *Computer* 36 (2003), Dec, Nr. 12, S. 68–75. http://dx.doi.org/10.1109/MC.2003.1250885. – DOI 10.1109/MC.2003.1250885. – ISSN 0018–9162

[18] FANG, Jianxin ; GUPTA, Saket ; KUMAR, Sanjay V. ; MARELLA, Sravan K. ; MISHRA, Vivek ; ZHOU, Pingqiang ; SAPATNEKAR, Sachin S.: Circuit Reliability: From Physics to Architectures. In: *Proceedings of the International Conference on Computer-Aided Design*. New York, NY, USA : ACM, 2012 (ICCAD '12). – ISBN 978–1–4503–1573–9, 243–246

[19] SHAN, Amar: Heterogeneous Processing: A Strategy for Augmenting Moore's Law. In: *Linux J.* 2006 (2006), Februar, Nr. 142, 7–. http://dl.acm.org/citation.cfm?id=1119128.1119135. – ISSN 1075–3583

[20] HERLIHY, Maurice: The Art of Multiprocessor Programming. In: *Proceedings of the Twenty-fifth Annual ACM Symposium on Principles of Distributed Computing*. New York, NY, USA : ACM, 2006 (PODC '06). – ISBN 1–59593–384–0, 1–2

[21] AMDAHL, Gene M.: Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In: *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*. New York, NY, USA : ACM, 1967 (AFIPS '67 (Spring)), 483–485

[22] NEWSOM, D. K. ; SERRES, O. ; AZARI, S. F. ; BADAWY, A. A. ; EL-GHAZAWI, T.: Energy Efficient Job Co-scheduling for High-Performance Parallel Computing Clusters. In: *2015 IEEE International Conference on Smart City/SocialCom/SustainCom (SmartCity)*, 2015, S. 550–556

[23] In: CHAPMAN, B. ; JOST, G. ; VAN DER PAS, R.: *Overview of OpenMP*. MITP, 2007. – ISBN 9780262255905

[24] GROPP, William ; LUSK, Ewing ; DOSS, Nathan ; SKJELLUM, Anthony: A high-performance, portable implementation of the MPI message passing interface standard. In: *Parallel Computing* 22 (1996), Nr. 6, 789 - 828. http://dx.doi.org/https://doi.org/10.1016/0167-8191(96)00024-5. – DOI https://doi.org/10.1016/0167–8191(96)00024–5. – ISSN 0167–8191

[25] KICHERER, Mario: *Reducing the Complexity of Heterogeneous Computing: A Unified Approach for Application Development and Runtime Optimization*, Karlsruhe Institute of Technology, Diss., 12 2014

[26] BECKER, Thomas ; YANG, Dai ; KÜSTNER, Tilman ; SCHULZ, Martin: Co-Scheduling in a Task-Based Programming Model. In: *COSH2018, Manchester, United Kingdom* (2018), January. http://dx.doi.org/10.14459/2018md1428536

[27] AUGONNET, Cédric ; THIBAULT, Samuel ; NAMYST, Raymond ; WACRENIER, Pierre-André: StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In: *Concurr. Comput. : Pract. Exper.* 23 (2011), Februar, Nr. 2, 187–198. http://dx.doi.org/10.1002/cpe.1631. – DOI 10.1002/cpe.1631. – ISSN 1532–0626

[28] SUN, Enqiang ; SCHAA, Dana ; BAGLEY, Richard ; RUBIN, Normman ; KAELI, David: Enabling Task-level Scheduling on Heterogeneous Platforms. In: *Proceedings of the 5th Annual Workshop on General Purpose Processing with Graphics Processing Units*. New York, NY, USA : ACM, 2012 (GPGPU-5). – ISBN 978–1–4503–1233–2, 84–93

[29] EVANS, Christopher: Notes on the opensurf library / University of Bristol. 2009. – Forschungsbericht

[30] REAÑO, C. ; SILLA, F. ; NIKOLOPOULOS, D. S. ; VARGHESE, B.: Intra-Node Memory Safe GPU Co-Scheduling. In: *IEEE Transactions on Parallel and Distributed Systems* 29 (2018), May, Nr. 5, S. 1089–1102. http://dx.doi.org/10.1109/TPDS.2017.2784428. – DOI 10.1109/TPDS.2017.2784428. – ISSN 1045–9219

[31] JIMENEZ, Victor J. ; VILANOVA, Lluis ; GELADO, Isaac ; GIL, Marisa ; FURSIN, Grigori ; NAVARRO, Nacho: Predictive Runtime Code Scheduling for Heterogeneous Architectures. In: *High Performance Embedded Architectures and Compilers*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2009. – ISBN 978–3–540–92990–1, S. 19–33

[32] SÜSS, Tim ; DÖRING, Nils ; GAD, Ramy ; NAGEL, Lars ; BRINKMANN, André ; FELD, Dustin ; SCHRICKER, Eric ; SODDEMANN, Thomas: Impact of the Scheduling Strategy in Heterogeneous Systems That Provide Co-Scheduling, 2016

[33] KRAUS, Jiri ; FÖRSTER, Malte ; BRANDES, Thomas ; SODDEMANN, Thomas: Using LAMA for efficient AMG on hybrid clusters. In: *Computer Science - Research and Development* 28 (2013), May, Nr. 2, 211–220. http://dx.doi.org/10.1007/s00450-012-0223-3. – DOI 10.1007/s00450–012–0223–3. – ISSN 1865–2042

[34] JIANG, Yunlian ; SHEN, Xipeng ; CHEN, Jie ; TRIPATHI, Rahul: Analysis and Approximation of Optimal Co-scheduling on Chip Multiprocessors. In: *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*. New York, NY, USA : ACM, 2008 (PACT '08). – ISBN 978–1–60558–282–5, 220–229

[35] JIANG, Y. ; TIAN, K. ; SHEN, X. ; ZHANG, J. ; CHEN, J. ; TRIPATHI, R.: The Complexity of Optimal Job Co-Scheduling on Chip Multiprocessors and Heuristics-Based Solutions. In: *IEEE Transactions on Parallel and Distributed Systems* 22 (2011), July, Nr. 7, S. 1192–1205. http://dx.doi.org/10.1109/TPDS.2010.193. – DOI 10.1109/TPDS.2010.193. – ISSN 1045–9219

[36] GAREY, Michael R. ; JOHNSON, David S.: *Computers and Intractability; A Guide to the Theory of NP-Completeness*. New York, NY, USA : W. H. Freeman & Co., 1990. – ISBN 0716710455

[37] TIAN, Kai ; JIANG, Yunlian ; SHEN, Xipeng: A Study on Optimally Co-scheduling Jobs of Different Lengths on Chip Multiprocessors. In: *Proceedings of the 6th ACM Conference on Computing Frontiers*. New York, NY, USA : ACM, 2009 (CF ’09). – ISBN 978–1–60558–413–3, 41–50

[38] NEELA MADHESWARI, A. ; WAHIDA BANU, R. S. D.: Co-scheduling of parallel jobs in clusters. In: *2009 2nd IEEE International Conference on Computer Science and Information Technology*, 2009, S. 71–75

[39] TOPCUOGLU, H. ; HARIRI, S. ; MIN-YOU WU: Performance-effective and low-complexity task scheduling for heterogeneous computing. In: *IEEE Transactions on Parallel and Distributed Systems* 13 (2002), March, Nr. 3, S. 260–274. http://dx.doi.org/10.1109/71.993206. – DOI 10.1109/71.993206

[40] VIKHAR, P. A.: Evolutionary algorithms: A critical review and its future prospects. In: *2016 International Conference on Global Trends in Signal Processing, Information Computing and Communication (ICGTSPICC)*, 2016, S. 261–265

[41] WEGENER, Ingo: Simulated Annealing Beats Metropolis in Combinatorial Optimization. In: *Electronic Colloquium on Computational Complexity*, 2004

[42] JUHÁR, J. ; VOKOROKOS, L.: Separation of concerns and concern granularity in source code. In: *2015 IEEE 13th International Scientific Conference on Informatics*, 2015, S. 139–144

[43] MULLER, Stefan K. ; WESTRICK, Sam ; ACAR, Umut A.: Fairness in Responsive Parallelism. In: *Proc. ACM Program. Lang.* 3 (2019), Juli, Nr. ICFP, 81:1–81:30. http://dx.doi.org/10.1145/3341685. – DOI 10.1145/3341685. – ISSN 2475–1421

[44] WONG, C. S. ; TAN, I. K. T. ; KUMARI, R. D. ; LAM, J. W. ; FUN, W.: Fairness and interactive performance of O(1) and CFS Linux kernel schedulers. In: *2008 International Symposium on Information Technology* Bd. 4, 2008, S. 1–8

[45] JOSE, J. ; SUJISHA, O. ; GILESH, M. ; BINDIMA, T.: On the Fairness of Linux O(1) Scheduler. In: *2014 5th International Conference on Intelligent Systems, Modelling and Simulation*, 2014, S. 668–674

[46] SALIU, Moshood ; SALAH, Khaled: Starvation Problem in CPU Scheduling For Multimedia Systems. (2019), 10

[47] LIU, Y. ; TANIGUCHI, I. ; TOMIYAMA, H. ; MENG, L.: List Scheduling Strategies for Task Graphs with Data Parallelism. In: *2013 First International Symposium on Computing and Networking*, 2013, S. 168–172

[48] WANG, J. ; LV, X. ; CHEN, X.: Comparative analysis of list scheduling algorithms on homogeneous multi-processors. In: *2016 8th IEEE International Conference on Communication Software and Networks (ICCSN)*, 2016, S. 708–713

[49] DHOTRE, S. ; PATIL, S.: Cause of process starvation for linux completely fair scheduler with apache server. In: *2017 International Conference on Computing Methodologies and Communication (ICCMC)*, 2017, S. 814–819

[50] DECHTER, Rina ; KASK, Kalev ; BIN, Eyal ; EMEK, Roy: Generating Random Solutions for Constraint Satisfaction Problems. In: *Eighteenth National Conference on Artificial Intelligence*. Menlo Park, CA, USA : American Association for Artificial Intelligence, 2002. – ISBN 0–262–51129–0, 15–21

[51] CHE, S. ; BOYER, M. ; MENG, J. ; TARJAN, D. ; SHEAFFER, J. W. ; LEE, S. ; SKADRON, K.: Rodinia: A benchmark suite for heterogeneous computing. In: *2009 IEEE International Symposium on Workload Characterization (IISWC)*, 2009, S. 44–54

[52] GOODRUM, Matthew A. ; TROTTER, Michael J. ; AKSEL, Alla ; ACTON, Scott T. ; SKADRON, Kevin: Parallelization of Particle Filter Algorithms. In: VARBANESCU, Ana L. (Hrsg.) ; MOLNOS, Anca (Hrsg.) ; NIEUWPOORT, Rob van (Hrsg.): *Computer Architecture*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2012. – ISBN 978–3–642–24322–6, S. 139–149