FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Adaptive Work-Stealing Techniques

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Lukas Haselsteiner
Matrikelnummer 0828683

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Jesper Larsson Träff
Mitwirkung: Martin Wimmer

Wien, 20.04.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK

Faculty of Informatics

Adaptive Work-Stealing Techniques

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Lukas Haselsteiner
Registration Number 0828683

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Jesper Larsson Träff
Assistance: Martin Wimmer

Vienna, 20.04.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Lukas Haselsteiner
Hernalser Hauptstraße 195/21, 1170 Wien

(Ort, Datum)	(Unterschrift Verfasser)

Acknowledgements

First of all I want to thank my advisors Jesper Larsson Träff and Martin Wimmer. Without their help - both in terms of technical insights and scientific writing - this work would not have been possible.

I would like to say a big thank you to Marlene and Lorenz for their advice in mathematical and statistical matters.

I want to thank my parents for their continuous support and always believing in me during my whole education. I also want to thank my sister who was always there for me when I needed her the most and who also reminded me to enjoy life outside university.

My time at university would not have been so much fun and I would have learned a lot less without a group of great colleagues and friends: Bernhard, Christoph, Martin, Roman and all the others - thank you for plenty interesting discussions and group works where I learned a lot.

I would like to thank my flatmates Carina, Johanna and Georg. We had an amazing time in our flat and they always cheered me up when things were not going the way I wanted them to go. They are "responsible" that Vienna got more than a second home for me and they supplied me with delicious food countless times!

Last but not least I want to thank all of my friends with whom I spent so much fun time during my study. I probably would have never finished my studies without this great group of guys!

Abstract

This thesis provides an in-depth discussion of improving the widely used work-stealing algorithm in situations where not all cores can be utilized. In the standard work-stealing algorithm, the number of active workers is kept fixed during the execution of the tasks. However, the parallelism of an application is seldom stable but often changes over the application's lifetime. This may decrease performance or waste resources, especially when multiple applications are running in parallel. When there are not enough tasks available to utilize all available processors, the unneeded workers waste processor resources and could slow down the whole application.

An approach to solve this problem is adaptive work-stealing. Adaptive work-stealing schedulers dynamically adjust the number of threads that are used to execute the tasks of an application. Such schedulers try to estimate the number of workers that can be utilized. Most of these schedulers split the execution of the application in so called quanta. Between two quanta it is checked how much time the workers spent stealing and how much time was spent working. Based on this classification workers are added or removed to process tasks in the next quantum.

A simple possibility to block workers that are currently under-utilized is the usage of a backoff. Backoffs are often used in computer networks to throttle senders when there exists a collision, because two or more computers try to send data at the same time. This work analyzes the usage of backoffs in work-stealing schedulers. We implemented and tested several different types of backoffs in the context of this work. We found out that it is crucial to select good parameter values for the backoffs, since some of them significantly influence the overall performance. The results are quite promising and our optimized backoffs improved the performance in our benchmarks notably. It was also interesting to see that in the end all different types of backoffs provide similar performance.

As a second main part, a new adaptive work-stealing algorithm is designed and implemented. It also uses the concept of quanta, but does not interrupt workers as long as they do not need to steal tasks. Therefore the approach has only a small overhead as long as there is enough work to process. We named our algorithm ”dynamic quanta algorithm” and tested the performance of the adaptive scheduler. Our algorithm performed better than the existing work-stealing scheduler, we compared our algorithms with. However, one surprising result is that in our benchmarks using an optimized backoff is at least as good.

Kurzfassung

Diese Arbeit beschäftigt sich mit Verbesserungen der oft genutzten Work-Stealing Technik in Situationen, wo nicht alle zur Verfügung stehenden Prozessorkerne ausgenutzt werden können. Im klassischen Work-Stealing Algorithmus ist die Anzahl der Worker-Threads während der Ausführung der Tasks fixiert. Jedoch ist der Grad der Parallelität einer Anwendung selten während der gesamten Laufzeit stabil, sondern ändert sich meist während der Lebenszeit der Anwendung. Diese Tatsache kann die Ausführungsgeschwindigkeit verringern oder Systemressourcen verschwenden, insbesondere wenn mehrere Anwendungen gleichzeitig auf einem System laufen. In Situationen, wo nicht genug Tasks vorhanden sind um alle verfügbaren Prozessoren auszunutzen, verschwenden nicht genutzte Worker-Threads Prozessorressourcen und können die Ausführungszeit der ganzen Anwendung vergrößern.

Ein Ansatz dieses Problem zu lösen ist adaptives Work-Stealing. Scheduler, die adaptives Work-Stealing nutzen, passen die Anzahl der Worker-Threads, die für die Ausführung der Tasks genutzt werden, zur Laufzeit dynamisch an. Solche Scheduler versuchen die Anzahl der Worker-Threads, die genutzt werden können, abzuschätzen. Die meisten dieser Scheduler teilen die Ausführung der Anwendung in so genannte Quanten auf. Zwischen zwei Quanten wird überprüft wieviel Zeit die Worker-Threads für das Stehlen von Tasks verwendet haben und wieviel für die tatsächliche Ausführung von Tasks. Anhand dieser Klassifikation werden Worker-Threads zur Anwendung hinzugefügt oder entfernt.

Eine einfache Technik, um Worker-Threads, die gerade nicht genutzt werden, zu pausieren, ist der Einsatz von sogenannten Backoffs. Häufig werden Backoffs im Netzwerkbereich verwendet: Wenn zwei Computer in einem Netzwerk gleichzeitig Daten versenden wollen, kann es zu einer Kollision der Daten kommen. In solchen Fällen werden oft Backoffs genutzt, um die Senderate der beteiligten Sendestationen zu verlangsamen. In dieser Arbeit wird untersucht, inwiefern sich dieses Konzept auch auf Work-Stealing Scheduler übertragen lässt. Im Rahmen der Arbeit wurden mehrere unterschiedliche Typen von Backoffs implementiert und getestet. Dabei hat sich herausgestellt, dass es sehr wichtig ist sorgfältige ausgewählte Werte für die Konfiguration der Backoffs zu verwenden. Die Testergebnisse unserer Backoffs sind vielversprechend: Durch die neuen Backoffs wurde die Leistung des bestehenden Schedulers in den durchgeführten Benchmarks bedeutend erhöht. Ein weiteres interessantes Ergebnis ist, dass alle getesteten Typen von Backoffs in etwa die gleiche Leistung liefern.

Im zweiten Hauptteil dieser Arbeit wird ein neuer adaptiver Work-Stealing Algorithmus entwickelt. Dieser Algorithmus verwendet ebenso das Konzept der Quanten, unterbricht Worker-Threads aber niemals, wenn sie selbst noch genug Tasks zum Ausführen haben. Deshalb verursacht dieser Ansatz nur einen sehr kleinen Mehraufwand, solange genug Arbeit vorhanden ist. Im Kontext der Diplomarbeit wurde auch die Performance des neuen ”Dynamic Quanta Algorithm” mittels Benchmarks getestet. Unser Algorithmus verbessert die Leistung des bestehenden Schedulers, mit welchem wir unsere neuen Ansätze verglichen haben. Dabei haben wir ein weiteres interessantes Ergebnis unserer Arbeit festgestellt: In unseren Benchmarks ist die Verwendung eines optimierten Backoffs mindestens genauso gut wie der adaptive Algorithmus.

Contents

CHAPTER 1

Introduction

Instead of significantly increasing the clock speed and instruction throughput of a single processor, processor vendors had been investing in multicore architectures for some years. This means that single-threaded applications hardly benefit from new processor generations [36].

Therefore programmers need to increase the concurrency level of an application in order to improve the performance. A widely used model is dividing the work into small tasks that can be executed concurrently. This model is also called *task-based (parallel) programming model* [33]. To efficiently process fine-grained tasks, a common practice is to start multiple worker threads that execute these tasks. The distribution of the tasks between workers is not trivial, often a work-stealing approach is used for this problem [12].

1.1 Motivation

Work-stealing is a decentralized scheduling mechanism that is used in many frameworks for (task-based) parallel programming. Among this frameworks are the widely used *Intel Threading Building Blocks* (TBB) [26] and Cilk [22]. In the original work-stealing algorithm, the number of active workers is fixed during the execution of the tasks [8]. However, the parallelism of an application is seldom stable but often changes over the application's lifetime. This may decrease performance or waste resources, especially when multiple applications are running in parallel.

In the last few years the idea of *adaptive work-stealing* has been presented by several authors [4, 15, 16, 19, 39]. In this thesis we use the term *adaptive* to describe schedulers that dynamically adjust the number of threads that are used to execute the tasks of an application. It is possible to use adaptive algorithms on many kind of different schedulers, but the focus of this thesis are adaptive work-stealing schedulers.

So far, the work in this area has been mainly focused on theoretical aspects of adaptive scheduling and we are not aware of any publicly available adaptive work-stealing scheduler. Additionally, we think in the area of adaptive work-stealing there is still enough room for new promising approaches that were not analyzed before.

In this work we analyze work-stealing schedulers with respect to properties that are relevant for adaptive work-stealing. One focus of the work is the usage of *backoffs* in work-stealing schedulers. Backoffs are a well-known technique that are often used in networks to reduce collisions on a shared media [1]. However, there hardly exists any work that analyzes the usage of backoffs in schedulers. Therefore we introduce and implement the most prominent backoff variants that are used in the area of networking [5, 10, 27] and analyze them in the context of work-stealing schedulers. Such backoffs can be configured with various parameters. We try to determine reasonable parameter ranges in the context of work-staling schedulers and optimize the parameters to get the best performance.

As a second main part, a new adaptive work-stealing algorithm is designed and implemented. Contrary to existing work, this algorithm can be used today by interested programmers without any modifications. As opposed to most approaches for adaptive work-stealing so far [4, 15, 39], we try to design a completely decentralized algorithm. We think this is a more general usable approach, since algorithms with a central entity likely have scalability problems on systems with a large number of available processing cores. The algorithms are implemented and integrated into the *Pheet framework* [40]. This allows us to evaluate performance and behavior of the adaptive algorithms with existing non-adaptive ones. For this evaluations micro benchmarks that are integrated in the Pheet framework are used.

1.2 Introduction of the Pheet Framework

We implement our algorithms in the *Pheet Framework* [40]. Pheet can be used like *Intel Threading Building Blocks* [26] or *Microsoft's Parallel Patterns Library* for implementing task parallel applications. However, due to its initial purpose as a library for research and teaching, it is highly customizable and extensible. Pheet is a C++ template library that makes use of many *modern C++ design principles* [6]. It comprises different centralized and decentralized schedulers, including a work-stealing scheduler which will be the foundation of our adaptive work-stealing scheduler. Nearly every component of the library can be exchanged or configured by the programmer using templates.

Additionally Pheet contains several micro benchmarks, which can be used to evaluate the performance of different schedulers or parameters of schedulers. For our needs the benchmark system is extended to be able to measure performance when more than one instance of the benchmark is run in parallel.

A more detailed description of Pheet is given in Section 2.2.

1.3 Structure of the Thesis

This work is structured as follows: In Chapter 2 we present existing work in the field of adaptive work-stealing. Beforehand we give an overview of work-stealing in general, and the Pheet framework in particular. Afterwards we focus on work about *adaptive* work-stealing. We analyze and compare the different approaches and determine which aspects we want to use in our own implementations and what properties we want to avoid.

Throughout the work we want to show the influence of our changes, improvements and new algorithms on the performance. For this purpose a set of micro benchmarks is executed on different machines. The benchmarks and systems used to test the performance are introduced in Chapter 3. The presented benchmarks are all part of the Pheet framework. We only had to implement a few extensions to the benchmark system to be able to benchmarks applications that are running concurrently at the same machine.

In the main part of our work we try to optimize the work-stealing scheduler of Pheet using adaptive techniques and we show novel algorithms for adaptive work-stealing. For each algorithm and optimization of the scheduler we present a chosen few benchmark results which we use to analyze the algorithm and for presentation of the pros and cons of the algorithms. This main part is split into two chapters.

Chapter 4 discusses the usage of backoffs in work-stealing schedulers. A backoff is a simple technique that can be used to put threads to sleep when there is not enough work available. So far backoffs are most often used in the area of networking and a lot of work exists that analyzes their usage in this area. We think that backoffs could also help to improve the performance of work-stealing schedulers. Therefore we implemented various different backoffs and show how they can be used and optimized for work-stealing. One surprising result of this work is that all types of different backoffs provide more or less the same performance improvements for work-stealing schedulers. However, it is important to choose reasonable values for the parameters that are used to control the behavior of the backoffs.

Chapter 5 presents our approach to adaptive work-stealing. We describe the algorithm and its important properties and show a couple of optimizations and extensions to the basic algorithm. Our algorithm does not use any centralized data structure, so it can scale also on machines with a lot of cores. Additionally it is designed in a way that the performance of applications that can utilize all available processor cores is not degraded.

Subsequently in Chapter 6 we give a comparison of the best or most interesting schedulers using the introduced micro benchmarks. The benchmarks show that both our backoffs and the adaptive algorithms can improve the performance of the original Pheet work-stealing scheduler. The tests also reveal that a work-stealing scheduler that uses a wisely tuned backoff often performs better than the scheduler that uses our adaptive work-stealing algorithm. In the end we state our conclusion and try to give an outlook of possible future work in the area of adaptive work-stealing in Chapter 7.

CHAPTER 2

State of the art

Adaptive work-stealing was first introduced by Agrawal et al. [4]. The term *adaptive* does not have a universal meaning in the context of scheduling. In this work, algorithms that dynamically adjust the number of threads that are used to execute the tasks of an application, are called *adaptive*. Based on the idea of Agrawal et al. several different approaches for adaptive work-stealing schedulers were developed.

Dynamically adjusting the number of workers executing the tasks of an application can be achieved by actually changing the number of workers of an application [4, 39] or by putting workers into a sleep state when they are not needed [15, 19]. In the case of *feedback-driven* adaptive work-stealing, workers provide some form of *parallelism feedback* to a central scheduler, which then assigns cores to the applications, based on this feedback. But there are also ways to achieve similar results in a decentralized manner.

This chapter first of all describes the basics of work-stealing in general. Afterwards several approaches to extend the classic work-stealing scheduler to an adaptive work-stealing scheduler are introduced. In the end of the chapter we give a summary and a comparison of the different schedulers. We also give an overview which ideas of the existing adaptive work-stealing schedulers we want to use in our own algorithm. More importantly we describe what properties of the existing approaches we want to improve or avoid completely.

2.1 Work-Stealing

Work-stealing is an often used decentralized scheduling mechanism. The concept goes at least back to Burton and Sleep's work on concurrent functional programs [14] and Halstead's implementation of Lisp on multiprocessors (Multilisp) [24]. Blumofe and Leiserson showed that work-stealing is provably time- and space-efficient [12]. The first non-blocking implementation of work-stealing was presented by Arora, Blumofe and Plaxton [8]. Due to the names of the authors, this algorithm is often referred to as *ABP* work-stealing. Arora et al. also provided time-bounds for work-stealing in multiprogrammed environments. In a multiprogrammed environment a parallelized application runs on a collection of processors that can grow or shrink during the execution. This happens regularly, since operating systems decide which application runs on which processors and dynamically adjust this assignment.

In work-stealing each thread (worker) manages a local deque (double-ended queue) of tasks that are ready for execution. Each time a new task is spawned, it is added to the end of the local queue. During execution workers dequeue tasks from the end of their local queue and execute them. When a thread runs out of work, it selects another worker (*victim*) and tries to *steal* a task from the victim's queue. The worker that tries to steal a task, is called a *thief*. A thief always tries to take tasks from the beginning of the queue. Therefore as long as the task queue contains more than one item, the other end of the queue is only accessed by one thread at a time. Often the end of the queue that can only be accessed by the owner of the queue is called *bottom*. The end where thieves try to steal tasks from is called *top*. If a steal attempt is unsuccessful, because the queue is empty or due to a spurious failure, the thief tries to steal from another victim. There are several possibilities how a thief selects the victim to steal a task. The strategy how to select victims is called *victim selection policy*. If the victim is chosen randomly, the scheduler is called *randomized work-stealing* scheduler. This randomization plays a prominent role in the proofs of the time-bounds of the work-stealing algorithm [8, 12]. However, in practice also victim selection policies that do not work completely randomized are used [40].

The execution of a task-parallel application is mostly modelled as a directed acyclic graph (dag). Each node represents an instruction and an edge represents a dependency between nodes. A node becomes ready to execute, when all predecessors of the node have been executed. The *work* T_1 of an application is the total number of nodes in the graph. The *span* or *critical path length* T_∞ is the length of the longest chain of dependencies. The *parallelism* of the application can be defined by $\frac{T_1}{T_\infty}$. It represents the average amount of work for each step in the critical path.

2.2 The Pheet Framework

Like described in the introduction we implement our algorithms in the *Pheet Framework* [40]. In this section some details about this framework that are important to understand our algorithms are described. Especially the differences to classic ABP work-stealing schedulers are outlined.

In Pheet a worker thread and its supporting data structures is called a *place*. A place is bound to a specific processing unit and is never migrated to another processing unit. Pheet makes use of a semi-random victim selection policy called *hierarchical victim selection*. For this victim selection policy the machine is modeled as a binary tree, where each place is a leaf in the tree. For the construction of the binary tree the communication costs between the places are taken into consideration: The communication costs are strictly non-decreasing with the distance in the binary tree. When the worker thread runs out of local tasks, it will select a victim out of all victims with same distance d by random. At first it is tried to find a victim with distance $d = 1$, and d is incremented each time a steal attempt fails. If a steal attempt is successful or the maximum distance in the hierarchy is reached, d is reset to 1. Such hierarchical victim selection policy can improve the data locality for the tasks and reduces the communication costs for stealing tasks.

In one steal operation it is usually better to steal half the work than just a single task [9]. But stealing more than one task also has some disadvantages. In Pheet between several possibilities to determine the number of tasks to steal in one steal operation can be chosen. In this work we use a scheduler of Pheet that utilizes an alternative approach for a good distribution of work between the workers: The scheduler always steals only one task, but before a new victim is searched using the victim selection policy, the thread attempts to steal a task from its last victim. If this attempt fails, a new victim is selected guided by the victim selection policy. When a steal attempt from a victim selected by this means fails, it is attempted to steal a task from this victim's last victim.

Pheet creates as many worker threads as processing units (processor cores) are used and pins each thread to one core. If only a subset of the available cores are used, which can be controlled by the user of the Pheet framework, only that many threads are created. However, this subset of threads is always pinned to the same processor cores in every application that uses Pheet. E.g., a programmer writes an application X that uses Pheet and configures Pheet in a way that at most four cores are used. Now two instances of this application X are run on an 8-core machine concurrently. Nevertheless only four cores of the machine are utilized, since Pheet pins the threads to the same four cores in both applications.

2.3 A-Steal

The A-Steal algorithm [4] is the first adaptive work-stealing scheduler. It is inspired by a centralized task-scheduling algorithm, called *A-Greedy* [3]. A-Steal assumes that the scheduling is done on two levels: On system-level a global *processor controller* exists that allocates disjoint sets of processors (or processor cores) to the applications. This means that the processor controller works as a space-sharing scheduler. For each program an application-level scheduler exists that executes the tasks on the allotted processors. A-Steal works as such an application-level scheduler. The application-level scheduler provides *parallelism feedback* to the processor controller. This feedback consists of the estimation of required processors of the application. So when an application cannot use all processors, the unneeded processors can be allotted to other applications.

The execution of an application is broken into equal-sized *scheduling quanta*. Each quantum has a quantum length L that defines the time span of a quantum. Before starting to execute a quantum q, the application-level scheduler estimates the application's need for processors (desire) d_q. This desire is reported as parallelism feedback to the processor controller. The processor controller determines the number of processors available for an application p_q (*processor availability*). The processor availability depends on the number of processors that is globally available and on the other applications that are currently running on the machine. In each quantum the number of processors an application gets allocated is the minimum of its desire and the processor availability $a_q = \min\{d_q, p_q\}$. This value is called the application's *allotment* a_q.

The time between two quanta is used for processor reallocation and other non-job related work. The execution of the tasks is interrupted in these periods of time. Choosing a suitable quantum length L is therefore important, since too short quanta would lead to a lot of overhead.

Since the symbols defined in the previous paragraphs are used quite often in the next sections, Table 2.1 shows an overview of the most important ones.

symbols	**meaning**
P	number of processors
q	quantum
L	length of a quantum
d_q	processor desire for quantum q
p_q	number of available processors in quantum q
a_q	number of allotted processors in quantum q

Table 2.1: Common abbreviation used in the following description of adaptive work-stealing

Compared to conventional work-stealing, an adaptive work-stealing scheduler has to deal with two new situations:

- *Allotment gain:* If an application receives more processors than in the last quantum, new worker threads can be used to process the tasks. Since the queues of these threads are empty, they immediately start stealing from other workers, like in classical work-stealing

- *Allotment loss:* On the other hand, the number of assigned processors can decline between two quanta. In such situations processors must be removed from the application, but they may have tasks in their queues, which still have to be executed. Such queues that are not attached to an active worker, are called *muggable*. When a worker thread runs out of work, instead of stealing immediately, it checks if there exists such a muggable queue. If there exists one, it *mugs* the queue, i.e. it takes the whole queue and continues work as if it was its own queue.

The most important part of the algorithm is the estimation of the processor desire d for the next quantum $q + 1$. For the estimation A-Steal tracks the *non-steal usage* n_q in each quantum. The non-steal usage is the number of cycles that were spent mugging or working in quantum q. At the end of a quantum, the non-steal usage is then used to classify the quantum by the following criteria:

- A quantum q can be either *satisfied* or *deprived*. It is called satisfied, if the application-level scheduler received as many processors as it had requested ($d_q = a_q$), otherwise it is classified as deprived.

- A quantum can also be *efficient* or *inefficient*. A *utilization parameter* δ is used, to distinguish efficient and inefficient quanta. A quantum q is *efficient* if $n_q \geq \delta L a_q$ and inefficient otherwise. So a quantum is efficient if at least a δ fraction of all cycles in it were non-steal cycles. The fraction $\frac{n_q}{a_q L}$ is also called utilization u_q. For this definition of ”efficient” the mug cycles are included. The reason for this is that mug cycles occur when the application loses one or more processors. So mug cycles do not indicate that too many processors were allotted to the application in general.

A-Steal uses a *responsiveness parameter* $\rho > 1$ to control how fast the processor desire is adjusted. The algorithm to estimate the desire for the next quantum, works the following way, depending on the classification of the last quantum:

- If quantum q was *inefficient*, the quantum contained many steal cycles. This means that the processors probably had an insufficient amount of work. Therefore the desire for the next quantum is decreased: $d_{q+1} = \frac{d_q}{\rho}$
- If the quantum was *efficient and satisfied*, the quantum contained only a small amount of steal cycles and got as many processors as requested. It is likely that the application can use more processors, and hence the desire is increased: $d_{q+1} = d_q\rho$
- If quantum q is classified as *efficient and deprived*, the quantum used all allotted processors efficiently, but it had requested more processors. However, it cannot be known if more processors could have been utilized, so the desire stays the same: $d_{q+1} = d_q$

The completion time of a set of tasks that can be executed in parallel, depends on the processor availability on the system. So the time bounds of A-Steal were analyzed relative to the processor availability. To make the results applicable in any environment, it is assumed that the system-level processor controller is an adversary to the application. However, even omniscient parallelism feedback schedulers cannot achieve good speedup relative to the mean processor availability P_{mean} against an adversary. An adversarial processor controller can make the processor availability large when the application's processor desire is small. Contrary when the application's desire is large, the controller can make the availability small. This means that P_{mean} is high, but the application is nevertheless almost always running serially.

The time bounds of A-Steal under these adversarial conditions were analyzed using *trim analysis*. Trim analysis is a technique used in the field of statistics, where a certain percentage of "outliers" (very high or low values) are discarded. To analyze the algorithm the *R-high-trimmed mean availability*, or just *R-trimmed availability* for short, was used by Agrawal et al. This R-trimmed availability is calculated in the following way:

- The processor availability of all execution steps is collected.
- The collected values are sorted by value.
- The largest R values are discarded. It is sufficient to trim the values just from the high side, trimming from the low side is not necessary.

Using this technique Agrawal et al. have shown that A-Steal completes the work in $O(T_1/\tilde{P} + T_\infty + L\,lg\,P)$ steps. Where $\tilde{P}$ is the $O(T_\infty + L\,lg\,P)$-trimmed availability; i.e. the mean processor availability except the $O(T_\infty + L\,lg\,P)$ steps with the highest availability. So a linear speedup - in respect to the trimmed availability - is achieved when T_1/T_∞ dominates $\tilde{P}$.

Additionally Agrawal et al. tested the empirical performance of the algorithm using several simulations. In these simulations A-Steal almost always provided a linear speedup with respect to the mean processor availability

2.4 Stable Adaptive Work-Stealing

Introduction

Cao et al. [15] propose another adaptive work-stealing scheduler, called *A-Deque*. A-Deque classifies quanta based on the processor utilization and the length of all task dequeues. Naturally, it is relevant if the processor is working or not (utilization) to determine the desire for the next quantum. The length of all queues gives the number of tasks that can be worked on, if enough processors are available. Therefore it is an indicator for the unexploited parallelism of the program.

In the following description of the algorithm $X_j(t)$ defines whether processor j at time t is working, mugging or stealing. If $X_j(t) = 0$ the processor is stealing, if $X_j(t) = 1$ the processor is working or mugging. The number of active queues at time t is denoted as e_t. This number includes the queues that are not attached to any worker and are waiting to be mugged. $Q_j(t)$ is the number of tasks that can be stolen from queue j at time t.

Additionally a so called *exploration parameter* $\beta \geq 0$ is used. It controls how aggressively the scheduler exploits the application's parallelism. When the exploration parameter is set to 1, for each task in a queue a new processor is requested. However, it can be assumed that often these waiting tasks spawn other tasks themselves. So setting the parameter to a value greater than 1 can be beneficial. If the extra processor is not needed, the desire will decrease quickly, since in such situations many steal cycles will occur.

Like in the A-Steal algorithm the quantum length L and the number of allotted processors a_q is taken into consideration. The processor desire for the next quantum d_{q+1} is calculated in the following way, where t_q denotes the time at the beginning of quantum q:

$$d_{q+1} = \frac{1}{L} \int_{t_q}^{t_q+L} \left(\underbrace{\sum_{j=1}^{a_q} X_j(t) + \beta \sum_{j=1}^{e_t} Q_j(t)}_{s} \right) \mathrm{d}t \qquad (2.1)$$

For the explanation of Equation 2.1 first of all we concentrate on the subexpression s between the braces: The first sum in s calculates the number of all processors working at time t. The second sum in s computes the number of tasks waiting in all queues at the same time t. This sum is then multiplied with the exploration parameter β. Since the whole subexpression s only describes the situation at a single point of time, the integral is used to take the whole quantum into consideration. Therefore the bounds of integration are t_q, the start of quantum q, and $t_q + L$, the end of this quantum. This value is divided by the length of the quantum L to get an average value over the whole quantum. This equation is of theoretical nature, a approximation that can be used in practice is described later in this section.

Implementation

A-Deque was implemented in *ACilk - Adaptive Cilk* as extension to the Cilk runtime system [22].

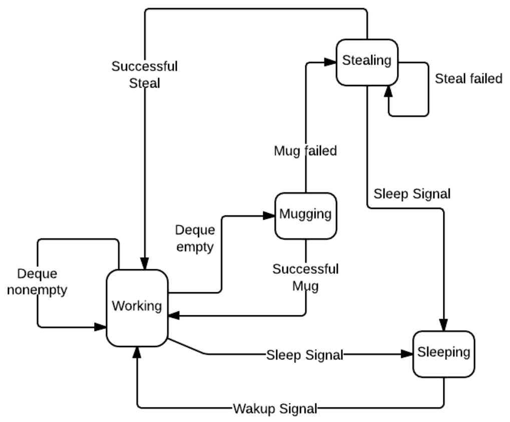


Figure 2.1: States of workers in the A-Deque algorithm

To support dynamic processor reallocation between applications during runtime, a global *processor controller* was implemented. It can be configured with quantum length, number of available processors and the parameters of the feedback algorithm. The controller works as a Linux daemon process between the operating system and the Cilk runtime. The programs report their processor desire to this daemon process, which uses the information to allocate the processors to the applications.

For each new application in ACilk, as many workers as there are physical processors are created. After the first allocation - usually 1 - the unneeded workers are put in a sleeping state. Besides sleeping, workers can have the states working, stealing or mugging. When the processor allotment of a program increases, the corresponding number of sleeping workers is woken up, whenever the allotment decreases, workers are put into the sleeping state. Figure 2.1 shows a state diagram of workers in the A-Deque algorithm.

The processor scheduling in the processor controller is done using the DEQ algorithm [29]. DEQ is a scheduling algorithm that divides the total number of processors equally between all applications, with the limitation that an application never receives more processors than it desired. Algorithm 2.1 shows the complete recursive procedure. At the beginning of every quantum the algorithm is called with a set $J(q)$ containing all active applications and the number of available processors P. In the pseudocode of the algorithm $d_i(q)$ is the processor desire and $a_i(q)$ the allotment of application J_i in quantum q.

Sampling methods for processor utilization and queue length

An important part of the A-Steal and A-Deque algorithms is measuring the processor utilization. A-Deque additionally requires the length of all active queues to calculate the processor desire. In practice collecting this information can severely impact the performance of the application. Thus efficient implementations of the algorithms that approximate the required statistics were developed [15].

Algorithm 2.1: DEQ(J(q), P) scheduling algorithm

```
if J(q) empty then
    return
end
S ←∅;
forall the j in J(q) do
    if $d_j(q) \leq P/|J(q)|$ then
        S ←{ S, j };
    end
end
if S empty then
    forall the j in J(q) do
        $a_j(q) \leftarrow P/|J(q)|$;
    end
else
    a ← 0;
    forall the j in S do
        $a_j(q) \leftarrow d_j(q)$;
        a ← a + $d_j(q)$ ;
    end
    DEQ(J(q) − S, $P$ − a) ;
end
```

To approximate the processor utilization, for each processor j in quantum q the following data is collected:

- Total number of *purely unsuccessful steal attempts* w_{jq}. A steal attempt is called purely unsuccessful if the victim itself is trying to steal from another worker, i.e. if the victim has run out of work itself.
- Total number of all steal attempts z_{jq}

The utilization u_q is therefore approximated with the following formula:

$$u_q = 1 - \frac{\sum_{j=1}^{a_q} w_{jq}}{\sum_{j=1}^{a_q} z_{jq}} \tag{2.2}$$

As a result $u_q a_q L$ can be used as an approximation for the term $\int_{t_q}^{t_q+L} \left(\sum_{j=1}^{a_q} X_j(t) \right) \mathrm{d}t$ of equation 2.1.

A processor is either working, mugging or stealing at any time, so this ratio gives a reasonable approximation for the *inefficiency* $1 - u_q$ of the processor in quantum q. Therefore it is an approximation to distinguish between efficient and inefficient quanta like described in the section about A-Steal. As long as a processor is only working, no additional overhead occurs during a quantum. However, between two quanta the number of (unsuccessful) steal attempts has to be collected from all workers. Some runtime systems like Cilk already use built-in counters to measure the number of steal attempts, so this approximation would incur just a little overhead.

To approximate the length of all queues in a quantum, at every steal attempt or successful mugging the length of the victim's queue is added up in a variable. This is done individually by every processor j and the accumulated value at the end of quantum q is denoted by ℓ_{jq}. For the approximation the number of active dequeues e at the end of quantum q is used. Since quantum q ends at time $t_q + L$ this number is denoted by e_{t_q+L} The approximated length of all active queues is therefore given by:

$$Q(q) = e_{t_q+L} \frac{\sum\limits_{j=1}^{a_q} \ell_{jq}}{\sum\limits_{j=1}^{a_q} z_{jq}} \tag{2.3}$$

The idea behind the formula is that the ratio between the added queue lengths and the number of steal attempts is the average length of a theoretic single queue in the quantum. Like in the approximation of the processor utilization, more steal attempts should lead to a better accuracy.

In addition to the average length of the dequeues during the quantum, the length of the queues at the end of the quantum is taken into consideration. For that, the length of the jth queue at the end of the quantum q is needed, which is given by $Q_j(t_{q+L})$

$$Q'(q) = \sum_{j=1}^{e_{t_q+L}} Q_j(t_{q+L}) \tag{2.4}$$

The two formulas are combined for the final approximation of the length of the active queues. It is given by:

$$\frac{1}{L} \int_{t_q}^{t_q+L} \sum_{j=1}^{e_t} Q_j(t)\, \mathrm{d}t = \alpha Q(q) + (1-\alpha) Q'(q) \tag{2.5}$$

The left part of this equation shows the theoretical average queue length of quantum q, like described for Equation 2.1. The right part is the approximation, which is a linear combination of $Q(q)$, the approximated length of all queues during the the quantum, and $Q'(q)$, the approximated length of all queues at the end of the quantum.

The more steal attempts occur during a quantum, the more accurate is the first approximation. Therefore it should have a higher weight in such situations. For this, the parameter α is used. It is suggested that this value is set to the ratio between the number of steal attempts and the maximum possible number of steal attempts. So when there are no steal attempts at all, the first approximation is not used altogether.

Combining formula 2.2 for the approximation of the processor utilization and equation 2.5 for the active queues length, leads to the final approximative calculation of A-Deque for the processor desire of the next quantum:

$$d_{q+1} = u_q a_q + \beta \left(\alpha Q(q) + (1-\alpha) Q'(q)\right) \tag{2.6}$$

To summarize, the first part of the equation $u_q a_q$ is an approximation of the utilization of all allotted processors in quantum q. The more unsuccessful steal attempts occur, the lower is this value. In the second part $Q(q)$ denotes the approximated length of all queues *during* quantum q and $Q'(q)$ is the length of all queues at the *end* of the quantum. $Q(q)$ and $Q'(q)$ are combined using $a \leq 1$. The exploration parameter β controls how aggressively the scheduler increases the parallelism, when there are tasks waiting in the dequeues.

Desire Stability

It is not so easy to estimate the processor desire in a stable way. Sun and Hsu [35] have analyzed the desire stability of A-Greedy [3], which uses similar concepts like A-Steal. They have shown that A-Greedy suffers from desire instability problems even when the parallelism is constant. Desire instability can degrade the performance of the application because of unnecessary context switches, loss of locality, etc. Therefore an algorithm with a stable desire is preferable.

Like A-Steal, A-Greedy uses a multiplicative-increase and -decrease strategy, to calculate the desire for the next quantum. Therefore desire instability can also occur when using A-Steal [15].

A simple piece of code is used as an example to show the calculated desire values of A-Steal and A-Deque. The sample application that is shown in Listing 2.1 spawns N tasks in a loop and afterwards it waits until these tasks are finished. It is assumed that the execution of these tasks take a long period of time and that no other tasks are created.

Listing 2.1: Sample code to show calculated processor desire values

```
for (i = 0,...,N-1)
  spawnTask
Wait For Tasks
```

In Table 2.2 the desire instability of A-Steal is shown for the sample application where N is set to 14. In an real world scenario hopefully more tasks are generated. However, for the description of the desire instability the magnitude of the number of tasks is not important. The responsiveness parameter ρ, which controls how fast the processor desire is adjusted, is set to 2. The utilization parameter δ, which is used to distinguish between efficient and inefficient quanta, is set to 0.9. It is assumed that the processor desire d_q is always fulfilled, i.e., the processor controller always allocates as many processors to the application as its desire is. Additionally it is assumed that there are only very few steal cycles during the execution. Therefore as long as there are fewer processors assigned to the application than spawned tasks exist, the quanta are categorized as efficient. As shown in Table 2.2, the number of processors fluctuates and will never converge to a stable state in this case

The given scenario is very simple, however it is quite clear that many parallel applications would suffer from desire instability in a similar way. For a specific situation, the issue can be improved with other ρ- and δ-values. Since parallelism can vary throughout an application, however, it is often not possible to find parameters that fit in each stage of the application.

Table 2.3 shows the estimated processor desires of the A-Deque algorithm, where again 14 tasks are spawned in total. The exploration parameter β, which controls how fast the processor desire is increased when tasks are waiting in the queues, is set to 2. In this example, one task

desire d_q	**utilization u_q**	**classification**
1	1	eff. and sat.
2	1	eff. and sat.
4	1	eff. and sat.
8	1	eff. and sat.
16	0.875	inefficient
8	1	eff. and sat.
16	0.875	inefficient
8	1	eff. and sat.

Table 2.2: Example of desire instability of A-Steal algorithm in an artificial scenario with 14 spawned tasks

desire d_q	**working cpus**	**tasks in queues**
1	1	1
3	3	1
5	5	1
7	7	1
9	9	1
11	11	1
13	13	1
15	14	0
14	14	0
14	14	0

Table 2.3: Example of growth of processor desire of A-Deque algorithm in an artificial scenario with 14 spawned tasks

is waiting for execution in all queues averaged over the entire quantum. Like in the A-Steal example, it is assumed that there are only a few steal cycles, as long as there are not more than N processors allotted. This is clearly a simplification. However, the important point is that after a finite number of steps A-Deque reaches a stable state. The desire does not change, since the parallelism is not changing, i.e. no tasks are waiting in the queues and the processors are utilized executing their respective task.

It can also be seen that A-Steal increases the parallelism, i.e., the processor desire, faster. It reaches the maximum level of parallelism in about $\log_\rho N$ steps. Whereas A-Deque is more conservative when requesting new processors and converges to the target parallelism in about $\frac{N}{\beta}$ steps. However, when the parallelism of the application does not change for some time, it uses the resources more efficiently and prevents unnecessary scheduling overhead like context switches and cache reloading. Nevertheless for large systems, $\frac{N}{\beta}$ steps can be a quite long time in which many processor cycles are wasted. Therefore this property influences the scalability of

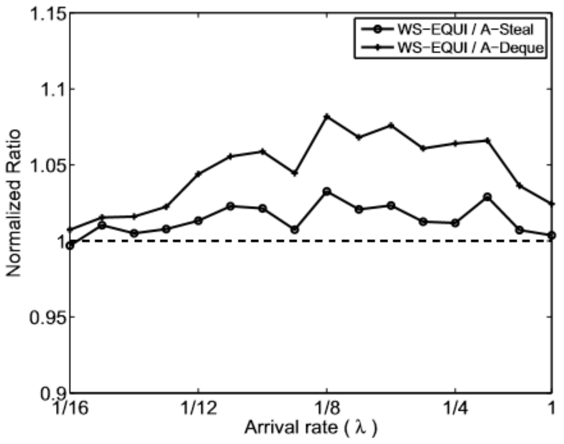


Figure 2.2: Performance comparison of different work-stealing algorithms with respect to makespan [15]

the A-Deque algorithm.

A-Steal and A-Deque Experiments

Using the modified Cilk-runtime ACilk and the approximations for the processor utilization and dequeue lengths, Cao et al. tested the performance of A-Steal and A-Deque. To compare the performance of A-Steal and A-Deque with an algorithm that does not make use of parallelism feedback, the *EQUI* algorithm [20] was used for the processor controller. The algorithm was named *WS-EQUI* in this context. It is very simple and equally shares the available processors among all running applications. When A-Steal and A-Deque are used as application-level schedulers, the presented *DEQ* algorithm is used, which takes the processor desire of the applications into consideration.

When running only one application, WS-EQUI performs better than A-Steal and A-Deque. This is because WS-EQUI is oblivious to the changes of an application's parallelism. No additional scheduling overhead occurs when the processor desire of a program changes, because it always allocates all available processors to the application. This result also corresponds with the observation that the original Cilk is more effective when running only one application on the system.

However, when multiple applications in parallel were run on the same machine, A-Deque and A-Steal in general performed better than the simple WS-EQUI scheduler. In the executed performance tests [15], A-Deque achieved better results in terms of makespan and response time. The term makespan defines the completion time of the last completed application when multiple applications are run in parallel. Response time is the duration a single application needs to complete. For the evaluation of the results the mean response time of all applications is used.

Figure 2.2 shows a comparison of A-Steal, A-Deque and WS-EQUI in terms of makespan. The figure shows the results as ratio of the makespan of WS-EQUI compared to A-Steal and A-Deque. Six different benchmarks were executed on an 8-core system. In total 16 application were started, which consisted of the six different benchmarks. The applications were not started at the same time, but according to a poisson process with an arrival rate between $\frac{1}{16}$ and 1. The system load is therefore proportional to the arrival rate of the applications. More details about the setup for the experiments can be found in [15]. The results show that in multiprogrammed environments both A-Steal and A-Deque outperform the simple WS-EQUI algorithm. However, the heavier the system load gets the less benefits adaptive work-stealing provides.

A-Deque benefits from its stable and more accurate parallelism feedback, compared to A-Steal. These results are quite promising and show that effective feedback mechanisms can improve the performance of applications with dynamic changing parallelism in multiprogrammed environments.

2.5 Palirria

Varisteas and Brorsson presented another scheduler for adaptive work-stealing, which they call *Palirria* [39]. They give more details and additional proofs in [38].

Palirria is an adaptive work-stealing scheduler that uses parallelism feedback like A-Steal and A-Deque. The estimation of the number of workers the application can actually utilize, is based on the length of the task queues. However, due to the usage of a deterministic victim selection policy *DVS* [39], only the task queues of a subset of the workers have to be checked. This enables an accurate estimation of processor desire with low overhead.

DVS replaces the randomized victim selection that is traditionally used in work-stealing. When using DVS two conditions must be met: Each worker thread has to be pinned to a different core and a metric for the communication distance between workers must be defined. This metric can either be mapped to the physical layout of the system or a virtual graph that contains all cores.

DVS is based on a theoretical model, where the processor model can have up to three dimensions. For clarity the description here is restricted to the case of two dimensions. To illustrate the concept of this metric, a simple mesh grid topology is used. In this model all nodes are connected to their neighbors horizontally and vertically and each node has at most four neighbors. The distance between two nodes is defined by the Manhattan distance between them, where the communication distance between adjacent nodes is 1.

A set of workers is assigned to each application and the execution is started on one of them. This worker is called *source* s. The rest of the workers is disjunctively divided into three different classes, which can also be seen in Figure 2.3:

- *class Z* includes all workers that have maximum communication distance from the source
- *class X* contains all workers that span horizontally and vertically from the source. Workers in class Z are not included in class X
- *class F* encloses all other workers

The idea of DVS is that the *flow of tasks* between workers happens in a predictable way. This flow automatically occurs, since during the lifetime of an application, tasks are stolen when

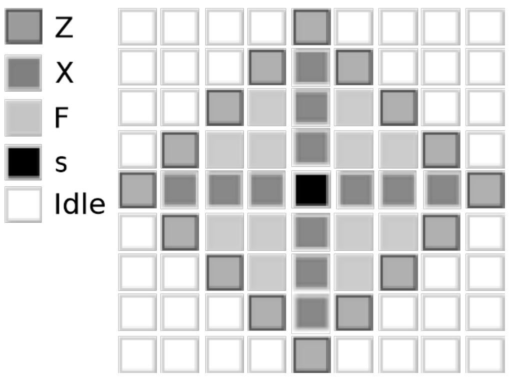


Figure 2.3: Worker classification according to the DVS rule set [39]

a worker runs out of work. To control this flow, each worker has a prioritized set of victims, from which it tries to steal tasks. Victims that contribute to a *primary flow path* are preferably selected. However, the set also contains other victims, since it is possible that no worker of the primary path is currently allotted to the application.

In summary, on the primary flow path stealing is only allowed between close neighbors (distance ≤ 2). Workers of class X transport tasks away from the source, workers in F relocate them back by primarily stealing from workers in class Z. Workers in Z mainly steal from other workers in the same class, to balance the load between all quadrants.

Using the classification of DVS, Palirria decides how the workers of the application are currently utilized:

- *Under-utilized* The size of all task queues of workers in class Z is 0; the allotment is *decreased.*
- *Over-utilized* The size of all task queues of workers in class X is above a *threshold* L; the allotment is *increased.*
- *Balanced* The size of all task queues of workers in class X is less than L or any task queue of a worker in Z is greater than 0; the allotment stays *unchanged.* One can also say that the utilization is balanced if it is neither under- nor over-utilized.

When the allotment of the application is changed, workers in one *zone* are added or removed. A zone is a subset of workers that are located at the same distance from the source. The members of a zone can be categorized into different classes. The maximum distance between the source and any worker in the current allotment is called *diaspora* d. When the size of the allotment is increased, workers of the outermost zone are removed. Upon increasing the size, workers at distance $d + 1$ are added to the allotment.

Due to the condition for decreasing the processor allotment, all workers that are removed from the allotment, have an empty task queue. However, it is possible that a worker that should be removed, is still working on a task. Furthermore new tasks can be spawned by this worker. A-Steal and A-Deque deal with this situation, by introducing the concept of *mugging* (see Section

2.3). In Palirria such situations are resolved in another way: Every worker that is removed continues to execute its current task. It is also allowed to add new tasks to its own task queue (task spawning) and it can be chosen as a victim. The worker exits automatically as soon as it runs out of work and its own task queue is empty. So a removed worker cannot steal tasks from other threads, but it still continues to work on its own tasks. The removal of a worker can also be revoked, when the allotment increases again, before the worker has finished processing all its tasks.

In the implementation of Palirria each application requires a *helper thread*. For all helper threads on the system one core is reserved. These threads run the processor desire calculations and add or remove workers when needed. So changes of the allotment run in the background with minimal interruption of the actual execution. However, the data that is needed for the decision algorithm is generated by the workers itself and thus affects their performance.

In Palirria the *quantum length* L corresponds with the interval in which the helper threads check the utilization state. It is the interval in which allotment changes can occur. Like in A-Steal and A-Deque this period is important for the performance: Running the procedure too often leads to high overhead due to the desire estimation calculation itself and may lead to unnecessary allotment changes. If the period is too long, workload changes may be missed and the parallelism of the workload may not be fully exploited.

Varisteas and Brorsson also performed experiments, to compare the performance of A-Steal, Palirria and a non-adaptive runtime in non-multiprogrammed scenarios. In both the implementations of A-Steal and Palirria, the scheduler adds and removes workers in sets (zones) as described above. Both adaptive implementations start with a set of five workers, i.e., the source and four workers in zone 1. The test runs were judged in terms of performance, wasted processor cycles and accuracy. Accuracy is the analogy between performance and resource utilization. So an estimation is accurate, when it is able to reduce the resource utilization without degrading performance. In average Palirria performed better than A-Steal in all three criteria. In comparison with the non-adaptive runtime, Palirria scored similar results in fine-grained highly parallel applications. However, the usage of Palirria was beneficial for irregular and coarser-grained programs. Comparisons with the A-Deque algorithm would have been quite interesting, since A-Deque uses similar criteria for resource utilization like Palirria. Unfortunately no such comparisons were made.

Since in Palirria tasks can only be stolen from close neighbors, it is unclear whether this algorithm is scalable to large systems. Additionally reserving one core exclusively for the helper threads of Palirria seems impractical. Especially when different kinds of applications are run on one system, where not all applications use Palirria for parallelization.

2.6 Balanced Work-Stealing

Ding et al. [19] introduced the *balanced work-stealing* (BWS) algorithm. The BWS scheduler aims at improving the behavior of work-stealing in multiprogrammed environments.

They introduced the concept that one can distinguish two different kinds of thieves: A *useful thief* is a thief that is stealing, when enough tasks are available. Contrary a *wasteful thief* is stealing, when there are no or only very few tasks available. Certainly the resources used by a wasteful thief, would be better used by busy workers or useful thieves. Many work-stealing schedulers try to mitigate this problem by calling the *yield* function, which common operating systems provide, on wasteful thieves. However, this behavior is not controllable: Applications that frequently yield cores tend to lose them to other applications, which can lead to significant unfairness. Additionally the overall performance can suffer, because the yielded core may be switched to another wasteful thief or back to the initiating thief too early.

BWS improves these issues by dynamically putting thieves to sleep or waking them up. Workers are put to sleep, depending on how hard it is for thieves to find a task to steal: When tasks that can be stolen can be easily found, it is decided that there are enough tasks to process. So sleeping workers are woken up. Contrary when it is hard to find tasks, it is assumed that there is insufficient work to execute and therefore workers are put to sleep.

Additionally, wasteful thieves do not yield their cores randomly, but they yield their cores directly to busy workers. When a thief fails to steal, it checks whether the victim has been preempted by the operating system and if it has an unfinished task to work on. If both is true, the thief directly yields its core to the victim, otherwise it tries to steal from another worker.

In comparison to the so far presented adaptive schedulers, BWS does not need any central data structure or worker. The data is collected in a decentralized manner and the scheduling decisions are directly made within the workers.

However, the two additions to the work-stealing scheduler require new features in operating systems that are currently not available in general:

- A possibility to check the running status of other workers is required.
- A possibility to yield a core directly to another worker.

A prototype of the BWS scheduler was implemented as a Cilk++ scheduler. The necessary operating system features were integrated into a modified Linux kernel. To test the performance in multiprogrammed environments, two applications where run in parallel in each benchmark run. Each run was performed with the original Cilk++ work-stealing scheduler and with the modified Cilk++ BWS scheduler. In average the BWS scheduler increased the overall system throughput compared to the original Cilk scheduler by 12.5%.

Ding et al. [19] also commented on the A-Steal algorithm. They claim that it can hardly be adopted on standard multicore-systems, where cores are time-shared between applications. In theory A-Steal assumes there is a space-sharing operating system, since the processor controller allocates disjoint sets of cores to different applications. It is furthermore noted that the quantum lengths in A-Steal must be quite long, to amortize the additional scheduling overhead. Yet parallel phases in applications can be rather short and so such phases cannot be exploited by the A-Steal scheduler.

2.7 Demand-aware Work-Stealing in Multi-programmed Multi-core Architectures

Chen et al. [16] presented another adaptive work-stealing scheduling technique called *demand-aware work-stealing* (DWS). The algorithm focuses on improving the performance of work-stealing applications in multiprogrammed environments. In the DWS algorithm applications use disjoint cores based on a space-sharing scheme. During runtime applications can release cores if they do not need all initially assigned cores or try to take cores that were released by other applications. The balancing of cores between co-running applications is achieved without a centralized OS-level core allocator.

Suppose on a system with n cores m applications are launched. The DWS scheduler starts one worker for each core in all the m applications. In every application n worker threads are running, like in the standard work-stealing algorithm. However, since a space-sharing scheme is used between all DWS applications, every application only gets $\frac{n}{m}$ cores actually assigned. All workers that are running on other cores are put so sleep. So on every core only one worker is running and contention between different worker threads of different applications is avoided. This scheme ensures fairness between the applications, but it is not adaptive and resources are wasted when the co-running applications can utilize different numbers of cores. Therefore in each application a so called *coordinator* thread is started that can communicate with the coordinators of other DWS applications. Using this coordinator an application can release cores if it does not need all assigned cores or can try to take released cores from other applications if it can utilize more cores than it has currently assigned. Releasing a core means putting the worker running on this core to sleep. When an application successfully takes a new core, accordingly the worker running on this core is woken up and continues to execute tasks. To support the dynamic reassignment of cores, the coordinators maintain a *core allocation table* in the shared memory. In this table it is stored which core is currently assigned to which DWS application.

In the DWS algorithm the decision to release a core is done on a local level for each worker. A core is released, if the number of consecutive failed steal attempts of the worker is above a threshold T_SLEEP. The authors of the paper suggest setting this threshold to n or $2n$ on an n-core machine. Whenever a core is released, the according entry in the core application table is set to 0 to signal other applications that this core is available.

The coordinator thread is responsible to check whether more cores can be used to execute the current set of tasks. In regular intervals the coordinator calculates how many tasks are waiting in the task queues of the application's workers N_b. The number of workers that should be woken up N_w is then calculated by the ratio of the queued tasks N_b and the number of active workers N_a:

$$N_w = \frac{N_b}{N_a} \tag{2.7}$$

The process of taking these additional cores further depends on the total number of free cores N_f on the machine. Additionally, it is relevant how many cores of the application's initial $\frac{n}{m}$ cores are taken by other applications. This number is denoted by N_r. Three different situations can occur:

$N_w \leq N_f$ When enough free cores are available, the application selects N_w random cores out of the free ones and takes them. The core allocation table is updated and the workers that are pinned to this cores are woken up.

$N_f \leq N_w \leq N_f + N_r$ In this situation there are not enough free cores available. But together with the N_r initial cores of the application that are currently used by other applications,

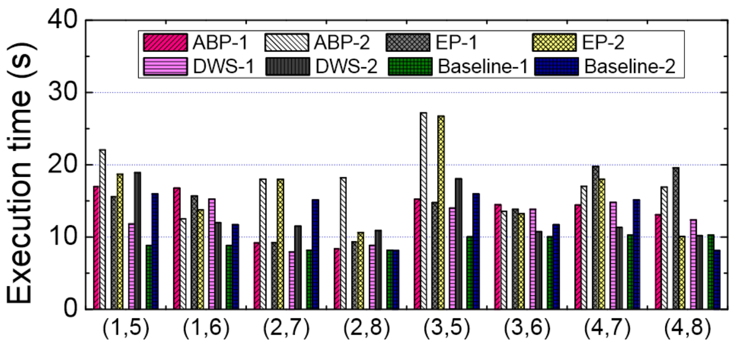


Figure 2.4: Performance of ABP, EP and DWS in benchmark mixes [16]

the application can nevertheless get its N_w additional cores. The application first uses all N_f free cores. Additionally it selects $N_w - N_f$ cores of the N_r cores that are currently taken by other applications and wakes the corresponding workers up.

$N_w > N_f + N_r$ In this case the application cannot use additional N_w cores, because there are not enough cores available. Therefore the application uses all available free cores N_f and takes back all N_r cores from other applications. In total it wakes up $N_f + N_r$ additional workers.

The process of taking back cores in the second and third case is not described in more detail. It remains unclear what happens with the applications where cores are taken away. It can be that worker on such a core continues execution until the application itself decides to put the worker to sleep. Like in Palirria the worker may continue execution but does not try to steal any tasks when it runs out of work. A third possibility would be to use the concept of mugging like in the A-Deque algorithm. However, the second and third possibility would mean that the coordinator needs to check if there are such cores that were taken back by another application.

A similar situation occurs when a new DWS application is started and other DWS applications are already running. The new application somehow should get an own set of cores, which must be taken from the running applications. However, such situations are not described in the paper.

The DWS algorithm was implemented by a modification of MIT Cilk [22]. For the implementation of the core allocation table a file is created which is mapped into shared memory using the Linux function `mmap`. After doing its work the coordinator thread goes to sleep for 10ms every time, to reduce the overhead of the thread. This principle is similar to the one used in the Palirria algorithm.

Chen et al. evaluated the performance of the DWS application on an 8-core machine with enabled Intel *Hyper-Threading Technology*. They treated the platform as a system with 16 cores, so each work-stealing program launches 16 workers on the platform by default. They evaluated the performance of the algorithm running two different benchmarks in parallel. Chen et el. compared their DWS algorithm with two other strategies for scheduling co-running programs: On the one hand they executed the programs using a standard ABP yielding mechanism (*ABP*). On the other hand they used an equi-partitioning policy, where the cores are evenly and statically allocated to the two co-running processes (*EP*). More details about the performance tests can be found in their work [16].

Figure 2.4 shows exemplary results of these tests. The x-axis denotes 8 different benchmark mixes. DWS-1 represents the execution time of the first benchmark, DWS-2 of the second benchmark, in the corresponding benchmark mix. The other bars represent the times for the corresponding task schedulers. The DWS algorithm performed up to 32% faster compared to the classic ABP work-stealing. It is also faster than the static equi-partitioning policy.

The DWS algorithm does not use a central scheduler for all applications, nevertheless there are two central components that can lead to contention: The core allocation table is accessed by all running DWS applications, which could be a bottleneck when a lot of applications are running at the same time. Especially since it is not described how concurrent accesses to this table are managed. Additionally the expense of the coordinator thread of every application is dependent on the number of available cores. The more cores are available on the system, the more workers have to be accessed to determine the number of tasks waiting in queues.

2.8 Insights for further work

Adaptive work-stealing and parallelism feedback are promising ideas to improve the performance of work-stealing in irregular parallel applications. Especially in multiprogrammed environments it can be very beneficial, to estimate the required resource usage to prevent wasting of resources. However, also scenarios where only one application runs on the system can profit from adaptive work-stealing. It can prevent unnecessary contention, when too few tasks to utilize all workers are available.

For fine-grained parallel programs parallelism feedback does not provide an advantage and may reduce performance. However, the presented algorithms show that with a careful implementation of adaptive work-stealing this disadvantage can be minimized.

Almost all presented algorithms were only tested very selectively, therefore additional tests are surely necessary to finally judge the algorithms. Since most of the ideas are quite new, there exist only few approaches that combine ideas of other papers. It may be beneficial to "cherry pick" the best parts of more than one approach and combine them into one algorithm.

The presented algorithms contain many interesting ideas and insights into adaptive work-stealing. However, we think that all of them have theoretical or practical drawbacks:

- Except BWS all algorithms rely on some kind of central entity that works as a dispatcher of threads. The problem with this kind of design is that the scalability is questionable when a large number of processor cores become available.
- The original *A-Steal* algorithm relies on counting of processor cycles. It is unclear how this can be implemented in practice with low overhead.
- In the *A-Deque* algorithm it takes $\frac{N}{\beta}$ quanta until the processor desire is increased to its maximum. This seems reasonable when there are not too many processors available. In really large systems a lot of resources can be wasted due to this property, especially when the concurrency of the application is fluctuating.

- Likewise the *Palirria* algorithm may need too long to distribute tasks to all processors on large systems, since tasks are only stolen from close neighbors.
- *A-Steal* and *A-Deque* interrupt all workers at the end of each quantum. In situation when enough tasks are available for all workers this may degrade performance.
- *BWS* uses operating system features that are not available on most systems today and therefore at the moment BWS is not usable in practice.

Nevertheless many ideas of the presented algorithms sound reasonable to use them in new algorithms:

- Adding or removing workers based on measurements of utilization for some timespan (a *quantum*), seems to be a good idea. Since the level of concurrency can be very fluctuating, measurements over the whole application lifetime do not make a lot of sense. Neither does measuring utilization at a single point of time sound like a promising idea.
- The classification of thieves as *wasteful* and *useful* seems reasonable and can be used for other adaptive work-stealing schedulers.
- We do not like the idea of removing a worker when it is still executing a tasks, which also requires to introduce the concept of mugging. In our view a better concept is used in Palirria, where ”removed” workers can finish executing their tasks, but are not allowed to steal tasks from other workers.

Based on this observations we try to design a new approach for adaptive work-stealing, using the ideas we like and trying to avoid the outlined disadvantages.

CHAPTER 3

Test Environments and Benchmarks

Throughout our work we want to show the influence of our changes, improvements and new algorithms on the performance. For this purpose a set of micro benchmarks, which are already part of Pheet, are executed on different machines. We will show exemplary benchmark results in the next two chapters when we think they are helpful, and give detailed benchmarks results afterwards in Chapter 6. In this chapter the used test environments and most important aspects of the benchmarks are described. A more detailed description of the benchmark suite can be found in Martin Wimmer's work [40].

3.1 Setup

For the benchmarks two shared-memory systems are used. One is a system consisting of 8 Intel Xeon E7-8850 processors. Each of the processors has 10 cores, so a total of 80 cores are available. Additionally simultaneous multi-threading could be used. Due to the fact that most applications are memory-bound by default Pheet does not take advantage of this. The system has 1TB of main memory, split into 8 NUMA nodes. Throughout this thesis, this system is called *Mars*. Figure 3.1 shows the processor and memory topology of Mars.

The other system comprises four AMD Opteron 6168 processors with 12 cores each, leading to a total of 48 cores. Each processor consists of two NUMA nodes with 16GB main memory each, so altogether 128GB of main memory is available. The machine is called *Saturn* and the detailed topology can be seen in Figure 3.2.

As compiler *GCC 4.8.2* is used setting the flag *-O3* to enable all standard compiler optimizations. The system on Mars runs with kernel version 3.14-1, whereas version 3.16-2 in use on Saturn.

For the benchmarks the default settings of Pheet are used, if not noted otherwise. This means that Pheet is configured *not* to use simultaneous multi-threading and the default behavior to pin the threads to the processing units is used.

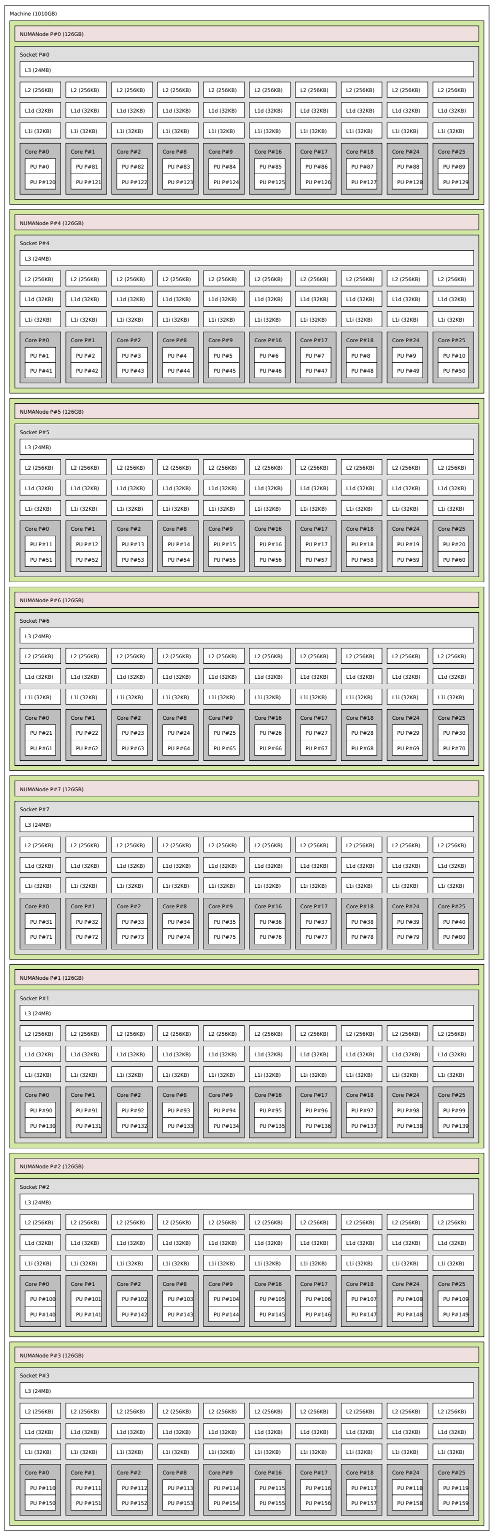


Figure 3.1: Processor and memory hierarchy of the *Mars* system generated with the `lstopo` utility of the *hwloc* library [13].

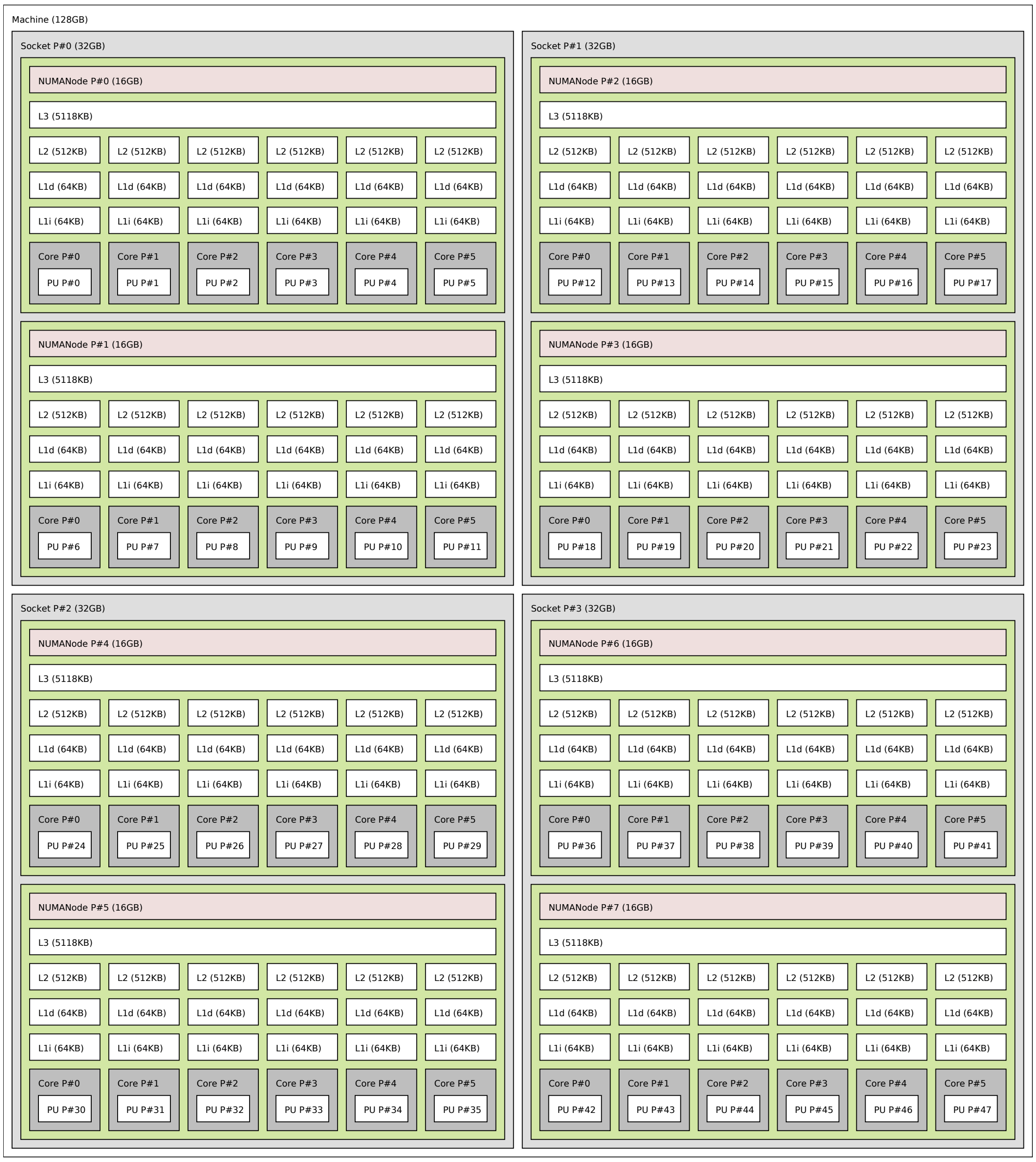


Figure 3.2: Processor and memory hierarchy of the *Saturn* system generated with the `lstopo` utility of the *hwloc* library [13].

3.2 Methodology

By default the results shown are mean values for 20 test runs. In some cases more or less than 20 experiments are performed, which is then explicitly mentioned in the text. In all graphs 95% confidence intervals are shown. The time needed to setup the test data is not included in the results.

For experiments where more than one instance of the benchmark is run in parallel, the mean execution time of all benchmarks executed in parallel is taken as a result for this iteration. Using an inter-process barrier the instances wait until all of them have finished initializing the test data. After the initialization phase the actual benchmark starts and the clock starts to run.

Comparison of benchmarks

When analyzing and comparing results of different benchmarks, one can ask which scheduler is the best one. However, this question is not easy to answer. Simply adding up the results of all different benchmarks favors the schedulers which perform well on the long running benchmarks, which is not intended. To make the results of the diverse benchmarks comparable, we first calculate the arithmetic mean of the benchmark results of all schedulers for each benchmark. The results of each scheduler are then divided by this mean value for every benchmark. This allows us to make a statement like 'Scheduler A performs twice as good in benchmark X compared to scheduler B, whereas B runs three times faster than A in benchmark Y.' Now the relative results of each benchmark can be summed up for every scheduler implementation and the results can be compared. To make this comparison easier, the summed up results are divided by the number of tested schedulers. A value of e.g., 2 therefore means that this scheduler performs two times slower than the average of all tested schedulers. We call the results we calculate using this method *aggregated results*. The absolute values in the aggregated results graphics are thus not meaningful and these graphics do not show the speedup that is gained when more cores are used.

We tried to choose a broad set of different problems, nevertheless, the results are still depending on the chosen benchmarks. Therefore we often show the results of each benchmark on its own, so that the reader can make his own conclusions.

3.3 Graph Bipartitioning

In the NP-hard [23] *graph bipartitioning problem* vertices of an undirected and weighted graph are to be partitioned into two sets. These two sets must have a given size and it is striven for a minimum total cut weight. The graph that is to be partitioned consists of n nodes and m edges. The problem is solved using the *branch-and-bound* paradigm. In each step one single node is assigned to one of the sets for each branch.

In general branch-and-bound algorithms are quite well suited for parallelization. The performance of such algorithms greatly depends on how soon good solutions are found, since they can be used to cut off branches that are not needed to be explored. Since in parallel implementations the order in which nodes are explored cannot be the same like in sequential algorithms so-called *speedup anomalies* can occur. These anomalies can be both negative and positive. When tasks that explore nodes that otherwise would not be needed to explore are created, the anomaly is negative. Contrary a task that finds a very good solution may be created very early and therefore branches can be cut off earlier than in a sequential implementation [17].

For our benchmarks we use dense random graphs of size $n = 35$ with an edge probability $p = 0.9$. The graphs are generated using the Erdős–Rényi method [21]. For every iteration of the experiment another random seed is used, but all benchmarks use the same random seeds. Since the variance between the iterations is quite large in this benchmark, 100 iterations instead of 20 were made for each scheduler.

3.4 LU decomposition

A LU decomposition (or factorization) algorithm factors a matrix as a product of two matrices: a lower (L) and an upper (U) triangular matrix. One important application of LU decomposition is solving systems of linear equations. It can also be used to calculate the inverse or determinant of a matrix.

For a LU decomposition to succeed, in some cases it is needed to reorder rows of the matrix. Algorithms that also take such cases into considerations are called *LU factorization with Partial Pivoting*. Since only rows are reordered, the method is called *partial pivoting*. The LU decomposition benchmark included in Pheet executes such a LU factorization with Partial Pivoting. Additionally to the *L* and *U* triangular matrices, a vector (pivots) is returned that describes the row permutations.

The parallelization of the algorithm is based on the following idea: The matrix can be split into several blocks. Instead of operating on scalars, the algorithm works on blocks, where each block itself is a matrix. This is beneficial since matrix-matrix calculations are more efficient than matrix-vector calculations and multiple blocks can be processed in parallel. Such an algorithm that works on blocks instead of single elements is called *algorithm-by-blocks* [34]. The *LAPACK* (Linear Algebra Package) library [7] is used for basic matrix and vector operations.

The benchmark operates on matrices of size $1024x1024$. The elements are generated randomly in the range $[-1, 1)$ and are of type double.

3.5 Prefix Sum

Prefix sum or *scan* is a primitive algorithm that is used for many more advanced algorithms like sorting, merging or calculating a minimum spanning tree [11]. The prefix sum algorithm works on an array and calculates for each item the sum of all items before this item.

The following two arrays show a sample input and output of the scan algorithm:

```
Input:  6 3 34  6  7  9  2  4
Output: 6 9 43 49 56 65 67 71
```

Calculating the prefix sums can be trivially achieved sequentially in a single pass. It consists of a single iteration over all elements of the array where all values are updated. On the contrary the parallel algorithm used in this benchmark works in two passes: In the first pass the array is split into blocks and the sum of all values in each block is calculated. Theses sums are then used in the second pass to calculate the actual prefix sums. Therefore the parallel version generates more work.

The benchmarks operate on arrays of size $n = 100,000,000$ where each element is initialized to 1. Initialization of all elements to 1 seems trivial at first glance, however the characteristics of the benchmark do not change when other values are used.

3.6 Quicksort

Quicksort is a well-known and often used sorting algorithm. Since quicksort uses the divide-and-conquer paradigm a basic parallelization of the algorithm is quite simple: The recursive calls of quicksort after each partitioning step are independent and can be executed in parallel. Since it is not efficient to sort only a few elements in parallel, the recursive calls are only made concurrently as long as the input array is longer than a selected cutoff value. Shorter arrays are than sorted sequentially.

However, using this parallelization scheme the partitioning step is still executed sequentially. In this step all elements of the input array have to be processed and therefore the critical path is of length $O(n)$.

Tsigas and Zhang [37] introduced a quicksort algorithm where also the partitioning phase is parallelized. In their algorithm the array is split into blocks and each thread takes one block from the left and one block from the right side of the array. Between these two blocks elements are swapped like in a classic partitioning step. After the elements are exchanged the thread takes a new block until all blocks are processed. Afterwards the remaining blocks and single elements are partitioned in a sequential way.

However, this algorithm that uses parallel partitioning, cannot be easily implemented using the standard task-parallel programming models. Pheet supports so-called *mixed-mode scheduling* which allows to implement a variant of this algorithm [41, 42]. Since this mode is no concern in this work we use the classic parallel version of the quicksort algorithm for our Quicksort experiments. However, this mixed-mode could also be used with the presented schedulers.

For the experiments arrays of size $n = 10,000,000$ are used where the elements are initialized using a gauss data distribution.

3.7 Single-source Shortest Path

Single-source shortest path (SSSP) is a problem where one has to find the shortest paths from one source node s to all other nodes in a graph. The problem is already extensively studied in sequential and parallel context [18, 31].

The benchmark included in Pheet is a quite simple parallelization of Dijkstra's algorithm. In Dijkstra's algorithm a tentative distance value is assigned to each node in the graph. In each iteration, beginning at the source, the node with the lowest distance value is *relaxed*. During this relaxation it is checked whether a neighboring node can be reached through the node that is currently relaxed with fewer costs. A priority queue is used to store the nodes that still need to be relaxed. This guarantees that each node is relaxed exactly once. After all nodes are relaxed, the shortest paths between the source and all other nodes are available.

In Pheet's parallel version of Dijkstra's algorithm multiple nodes are relaxed in parallel. Each node is relaxed in an own task. Therefore *relaxing* a *node* means *executing* a *task* in the parallel version. Due to the parallel execution of the relaxation, some nodes might be relaxed too early before their distance value is fixed. Therefore it happens that some nodes need to be *re-relaxed*. So the original relaxation of such nodes is useless work.

The prioritization which nodes are relaxed next corresponds with the prioritization which tasks are executed next by the scheduler. Pheet offers scheduling strategies that allow to assign priorities to tasks and so the order in which tasks are executed can be controlled. This is used by the parallel version of Dijkstra's algorithm and therefore replaces the priority queue of the sequential version. It can happen that a better path for a node is found, when there is still another task for this node in the queue. In such cases nevertheless a new task is executed and the scheduler at some point discards the old task.

The graphs for the benchmarks are again generated using the Erdős–Rényi method. The graphs have $n = 10,000$ nodes and an edge probability $p = 0.5$.

3.8 Unbalanced Tree Search

Unbalanced tree search (UTS) is a benchmark that was designed to measure performance of parallel applications that require dynamic load balancing [32]. In the benchmark the number of nodes of a given tree must be counted. The traversed tree is generated dynamically. Each node has a descriptor, which is used to determine the number of children of the node. The child node descriptors are generated using a cryptographic hash on the combination of parent descriptor and the index of the generated child node. The node descriptors also work as a random variable that determines the number of children of the node.

To determine the overall shape of the generated tree one of two different tree types can be selected. The tree type determines the probability distribution that is used to generate the children of the node. Either a *geometric* or *binomial* distribution can be used. Accordingly a *geometric* or a *binomial* tree is produced.

In a binomial tree each node has m children with a probability q and no children with probability $1 - q$. Therefore for the generation of binomial trees the parameters m and q have to be chosen. A binomial tree is a perfect adversary for load balancing systems, since the expected work is the same at all nodes. This implies that knowing where a node is located in the tree, does not help to estimate the size of the subtree.

For geometric trees the parameter d limits the maximum depth of the generated tree. In geometric trees the expected size of a subtree increases the closer the root node of the subtree is located to the root. This kind of generation yields very unbalanced trees.

For each node in the tree a task is spawned and executed by the scheduler. Such tasks are very lightweight and therefore this benchmark is well-suited to show and compare the overhead of different schedulers.

We use three different Unbalanced Tree Search benchmarks. The numbers used in the names of these three benchmarks coincide with the parameter used to select the corresponding variant in the Pheet benchmark suite:

UTS-0 In this benchmark a geometric tree with a maximum depth $d = 10$ is generated, which results in a total of $4,130,071$ nodes.

UTS-4 In the second unbalanced tree search benchmark a hybrid tree with a maximum depth of $d = 16$ is used. Hybrid trees are generated with geometric distributions near the root and binomial distributions towards the leaves. For the binomial parts of the tree the parameter values $q = 0.234375$ and $m = 4$ are used. In the end $4,132,453$ nodes are included in this tree.

UTS-7 In the third UTS benchmark a binomial tree with $r = 5$ and $m = 5$ is traversed, which consists of $111,345,631$ nodes.

3.9 Variety of Benchmarks and Systems

We tried to choose a set of benchmarks that test different aspects of the task scheduler. Some of the benchmarks are computation bound (e.g. graph bipartitioning and UTS) and others are memory bound (e.g. LU decomposition and Quicksort). Not all benchmarks can utilize all available cores on this large machines, whereas others benefit from the usage of (almost) all cores. However, this is also true in real-life applications, where it is not always possible to split the work in enough parts to profit from to usage of lots of processors. Nevertheless six benchmarks can only show some performance characteristics of the scheduler and it is not possible to show results which can be generalized to all different types of applications. Therefore it is always a good idea to run specialized benchmarks when the performance of an application is critical.

We tested our schedulers on two different systems. Although the benchmark results show that these systems sometimes behave rather differently, we are well aware that tests on only two machines cannot be used to judge a scheduler or algorithm conclusively. On one hand exhaustive testing on a system takes a lot of time and on the other hand sometimes other problems stand in the way of running benchmarks on other machines. We would have liked to show results on more machines. Pheet uses standard C++ and only common libraries, which are good preconditions to execute tests on a variety of different systems. However, we ran into practical problems like incomplete C++11 support, which prevented tests on more systems.

CHAPTER 4

Backoff Techniques

The original ABP work-stealing algorithm uses a `yield` system call between every pair of consecutive steal attempts [8]. Pheet's work-stealing scheduler uses a slightly different approach: It does not use `yield`, but a so called *backoff*. A backoff is an algorithm that is used to slow down some process, until a suitable process rate is found. In case of Pheet this means that every time a thread *backs off* it is blocked for some timespan. The backoff time is increased every time a thread calls the `backoff` function. How much the time is increased depends on the specific implementation.

4.1 Backoffs in other fields

Backoffs were first used in *Aloha*, which is a wireless, packet-switching network [1]. The backoff was adapted for the *Ethernet* and is used in various synchronization techniques on shared memory multiprocessor systems [2, 30]. A backoff can also be used in locks to prevent unnecessary contention when two or more threads try to access the critical section at the same time [25].

There exist several variants of backoff mechanisms that can be distinguished by their way to determine the time to sleep:

Linear backoff Each time the process or thread backs off, the time to sleep is increased by a constant value [2, 30].

Exponential backoff When an exponential backoff is used, the time to back off is increased by multiplying it with a backoff factor $r > 1$. In the special case of $r = 2$, the backoff is called *binary exponential backoff* (BEB) [2, 25, 28].

Proportional backoff A proportional backoff can be used, when some kind of information that helps to estimate the time to wait, is available. The process is than delayed proportional to this estimated time. E.g. such a backoff can be used for a *ticket lock*: Each thread knows its own ticket number and the number of the ticket that is currently served. Since threads that have a higher ticket number need to wait longer to enter the critical section, such threads can be put longer to sleep than threads with a lower number. Therefore a proportional backoff can be used. In such cases other types of backoffs can be disadvantageous, since a backed off thread with a small ticket number can stall other threads with higher numbers [30].

Exponential-linear backoff The *exponential-linear backoff algorithm* (ELBA) is a technique proposed for contention-based wireless networks. It is a combination of a linear and an exponential backoff: As long as there are few transmission collisions the backoff time is increased in an exponential way. However, when some threshold is exceeded, the backoff increases the time in a linear way [28].

Beside this general classification of backoffs, in each backoff type the following properties can be customized:

Randomness To prevent that two threads coincidentally wake up at the same moment, some randomness can be added to the calculation of the backoff time. On the one hand this can be achieved by adding some random value or by multiplication with a random factor. On the other hand some implementations do not use the calculated backoff time directly to put the thread to sleep, but use it as a maximum value for the sleep time. The actual sleep time is a random value between zero and the current backoff time.

Decrease of backoff time It can be decided how the backoff time is decreased after a successful attempt. The backoff time can simply be reset to its minimum value, the same strategy for increasing the time can be used for decreasing it or a completely different method can be applied.

Minimum and maximum backoff time The minimum backoff time ideally is the time the thread has at least to wait before it can continue its work. The maximum is important to prevent threads from waiting a very long time, which probably leads to an unnecessary delay of work. Backoffs that make use of a maximum backoff time are also called *truncated* [27].

In the area of networking many studies about the performance and especially about the stability of exponential backoff algorithms exist [5, 10, 27]. However, Kwak et al. state that these studies often produced contradictory results on the stability of exponential backoffs, due to various used definitions of stability and due to modified or simplified models of the backoff algorithm. The following two groups of stability definitions exist [27]:

Throughput definition Using the throughput definition the backoff algorithm is stable when the throughput does not collapse if the load goes to infinity.

Delay definition Under the delay definition the algorithm is stable when the waiting time is bounded.

These contrary definitions make it hard to give a summary of the existing papers. Additionally most of the existing analyses regarding performance and stability of backoffs are done in the context of a specific network medium access protocol like Ethernet or WLAN. This means that there is one shared medium that is tried to accessed and where collisions need to be prevented. This is quite different to the usage of backoffs in work-stealing schedulers, where there is neither a shared medium that is tried to accessed nor do collisions exist. Since we could not think of a good analogy between networks and scheduling, we did not research the existing work on the usage of backoffs in networks any deeper.

4.2 Backoffs in work-stealing schedulers

The idea of using a backoff mechanism for a (work-stealing) scheduler is not new, but there exists hardly any deeper analysis. We think that this mechanism can also be seen as a simple form of an adaptive scheduling technique. Therefore we want to analyze to which extent such backoffs can help to improve the performance of a work-stealing scheduler.

By default Pheet uses a *truncated exponential backoff*, where the time is doubled whenever a thread backs off. Pheet does not back off after all unsuccessful steal attempts, but this depends on the hierarchical victim selection described in Section 2.2. As long as the maximum distance d to find a victim can be increased, the backoff is not used. Only when the maximum distance in the hierarchy is reached, before the distance is reset to 1 again, the worker thread backs off.

In Pheet the minimum default time for the backoff is 100 nanoseconds and the maximum time is 0.1 milliseconds. The thread is put to sleep for a random timespan between zero and the calculated backoff time. When a task to steal is found, the backoff is reset to its minimum time. All benchmarks that refer to the *original backoff* are made using these values, unless noted otherwise.

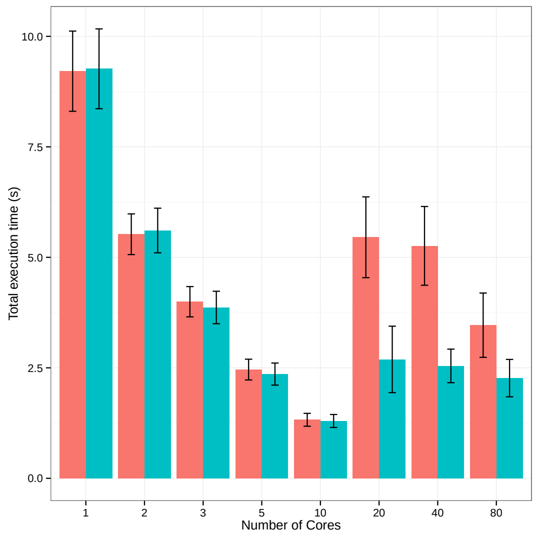
In the following pages we try to give an answer to the following questions:

Is a backoff useful for a work-stealing scheduler? For this question the performance of schedulers with different backoff variants are compared against a scheduler that uses the classic approach with yield calls and against one that neither uses a backoff nor yield calls.

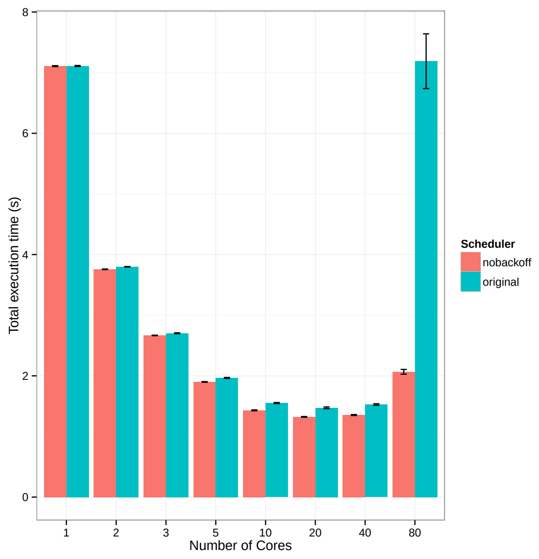
How do different backoff variants perform? In the previous section several different variants of backoffs were introduced. We want to give an overview how these alternatives perform and present our findings regarding sensible parameters of the backoffs.

Is there a difference between various applications regarding the usage of backoffs? It is conceivable that the performance of the backoffs differs depending on the actual application. Using the Pheet micro benchmarks we test if and what differences exist.

Are there differences between dedicated and multiprogrammed environments? Often performance analyses are focused on dedicated environments where only one benchmark is run at a time. We want to find out how different backoff variants perform in multiprogrammed environments and if there are differences compared to dedicated environments.



(a) Graph Bipartitioning Benchmark



(b) LU decomposition benchmark

Figure 4.1: Graph Bipartitioning and LU factorization benchmarks on Mars that show two contrary performance characteristics regarding the usage of a backoff

4.3 Analysis of existing scheduler

Before working on improvements of the scheduler, we give an overview of the performance of the existing scheduler. We especially focus on the aspect if and how the used backoff mechanism improves the performance.

Figure 4.1 shows the results of two benchmarks that were run on Mars in a dedicated environment. One time the benchmark is run with the original scheduler with enabled backoff (original), one time the backoff mechanism is completely disabled (nobackoff). In Figure 4.1a the results of the Graph Bipartitioning benchmark are shown. When 20 and more cores are used, the benchmark clearly benefits from the usage of a backoff compared to completely disabling the backoff. In Figure 4.1b, which shows the results of the LU decomposition benchmark, an opposite effect can be seen: The usage of the backoff more than triples the time to finish the benchmark when all 80 cores are used.

Additionally one can see a so far not mentioned aspect of highly parallel machines: Not all applications benefit from the usage of a vast number of cores. On the contrary, the performance of applications that are not parallel enough, can even degrade when the work-stealing scheduler tries to utilize too many cores. We use the term *slowdown* to describe this phenomenon. Since schedulers often use all available cores by default, in practice this is an issue for many applications. The developer of an application could give the scheduler hints about how many cores should be used. However, this is seldom possible since this depends on the actual machine and can depend on the size of some kind of input data.

The scheduler using the backoff mechanism seems to run into the problem that it is sometimes too eager putting workers to sleep. If the degree of parallelism is varying, the backoff

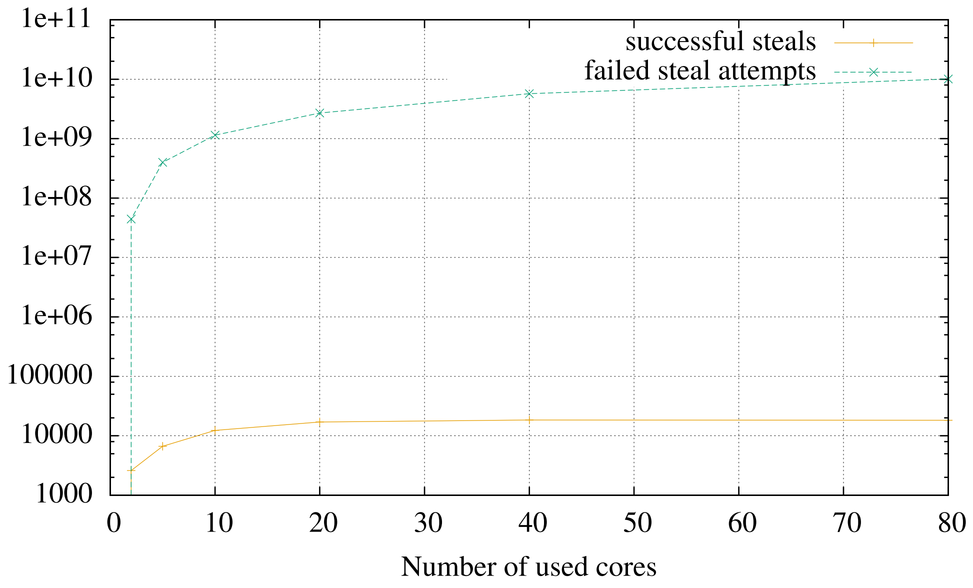


Figure 4.2: Number of steals in LU decomposition benchmark on Mars without using a backoff

mechanism may put workers to sleep for too long and may miss periods with a high degree of parallelism. This seems to be the case in the shown LU factorization benchmark.

If the scheduler would always perform worse with enabled backoff, the natural corollary would be to disable it in general. However, other application types like the graph bipartitioning benchmark show a different behavior.

These results show that without any additional information, we cannot know if an arbitrary set of tasks is processed faster by the scheduler with enabled backoff or by the one with disabled backoff. Since Pheet is highly customizable, the backoff can be enabled or disabled by the programmer, if he knows beforehand which scheduler is advantageous.

However, if this decision cannot be made at compile time, adaptive scheduling may accommodate the behavior at runtime in a similar manner. Additionally we try to implement an optimized backoff that does not have such bad worst-case scenarios like the backoff currently used in Pheet. The benchmark results in Chapter 6 show that both approaches achieve similar better results.

One important insight of these tests is that our adaptive scheduler at least needs to be compared to the existing scheduler with backoff both enabled and disabled. If the new algorithm in each benchmark performs roughly as good as the better of the two existing ones, this would already be a very nice result. However, this may be very tricky, since it seems like that this is highly dependent on the executed set of tasks.

Analysis of steal attempts

One possibility to analyze a work-stealing scheduler is to look at the number of steal attempts and the number of successful steals. Figure 4.2 shows these two numbers for the LU decomposition

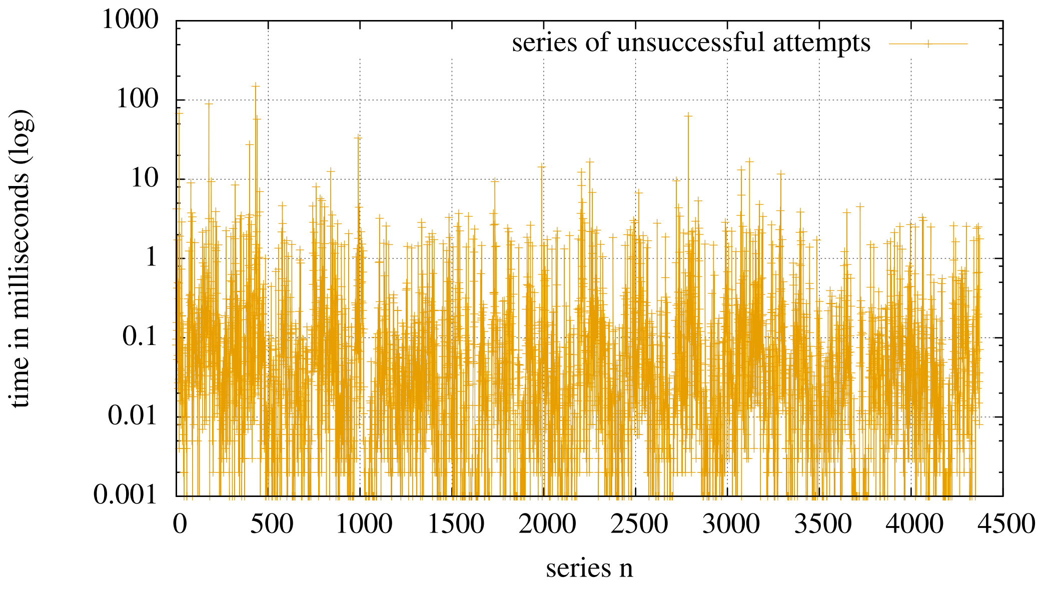


Figure 4.3: Duration of series of unsuccessful steal attempts in LU decomposition benchmark on Mars

benchmark. Since there are far more failed steal attempts than successful ones, the graph has a log scale y-axis. The graph shows that the number of steal attempts is steadily increasing the more cores are utilized. This is neither surprising nor worrisome, since in many applications the majority of cores has to steal work from other cores now and then. Therefore the more cores are used, the more steal attempts occur. However, in this figure one can also see that the number of successful steals is significantly increasing only until 20 cores. The number of successful steals is only slightly higher for 40 cores and does not increase when 80 cores are used. This is an indicator that there is simply not enough work to utilize all processor cores. However, on all cores workers are running that try to steal work from other workers.

To prevent such bad effects, it would be beneficial not to utilize all available cores. In an ideal scenario, we would be able to recognize beforehand or after a short period of time, e.g., that it is better just to use 40 cores and not 80 cores. Quite apart from the fact that this would be very hard on a global level, it is even more difficult to make such decision on a local level for every worker. We need to identify characteristics on a per-worker level which indicate that it is beneficial to put the worker to sleep.

The behavior of a work-stealing system can also be analyzed from the following point of view: The best utilization is achieved, when each worker has enough local tasks and does not need to steal tasks from other threads. When a worker runs out of tasks, preferably it does not need many steal attempts to find a new task to work on. However, if there is not enough work, many consecutive steal attempts can occur before the next successful one. Such *series of unsuccessful steal attempts* could be a good starting point for putting a worker to sleep. Instead of permanently trying to steal tasks wasting processor cycles, it could also be put to sleep.

Figure 4.3 shows the duration of such series of unsuccessful steal attempts of one worker, when the backoff is completely disabled. The data was gathered during a run of the LU decomposition benchmark on Mars. Every time the first of a series of unsuccessful steal attempts occurs, a timer is started. This timer is stopped after a successful steal. Therefore each point in the diagram shows the time between the first unsuccessful steal attempt of a series and the next successful one.

One can see that most of the time a task to steal is found in less than one millisecond. Compared to the vast majority of cases where a task is found very quickly, there are just a few cases where it takes around one millisecond to find a new task. In very few cases it even takes around 10 milliseconds. Since microsecond accuracy is typical for modern systems, threads can actually be put to sleep for time periods of around one millisecond. Additionally this graph does not show if the worker just finds a single task to steal in regular intervals but nevertheless spends most time with unsuccessful steal attempts. For such threads it could also be a good idea to introduce longer periods of sleep.

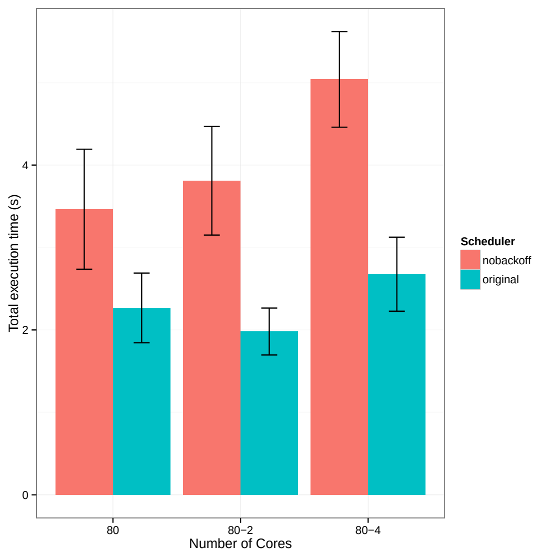
The gathered data provides some insights into the characteristics of the work-stealing scheduler. However, the collected data is not really conclusive how and if an adaptive scheduler can improve the situation. It seems like just counting the number of unsuccessful steal attempts or measuring the time these series of unsuccessful attempts take, would not be enough. Additionally it must be said that collecting the data itself changes the behavior and performance of the system.

Executing multiple instances in parallel

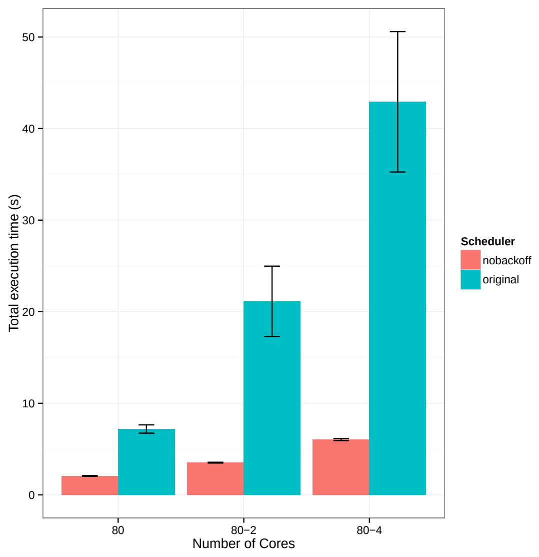
When multiple applications that compete for processor resources run in parallel, adaptive scheduling promises even more improvements. To analyze such situations the benchmarks were executed in the following way: The benchmark instances are started and each instance immediately starts with the single-threaded initialization of its test data. However, after the test data is initialized, each instance waits until all other instances are finished with their initialization using a process barrier. So all instances start the multi-threaded benchmark at the same time.

Tests were executed running one, two and four instances of the benchmarks in parallel, as seen in Figure 4.4. The results labeled *80-2* and *80-4* show the average execution time of the benchmarks when 2 and 4 instances are run in parallel. Since both benchmarks cannot utilize all cores, one might expect that it is better to run more than one instance in parallel. It would be possible that each instance on average takes less time than running only one instance. Again the tests show contrary results for the two benchmarks: In Figure 4.4a the graph bipartitioning benchmark results are shown. The average execution time stays roughly the same, even when two or four instances of the application are run in parallel. The LU decomposition benchmark, which is shown in Figure 4.4b behaves differently and the performance gets notably worse when more than one instance is run in parallel. The significant result is that the scheduler with enabled backoff performs awfully badly for the LU factorization benchmark. This coincides with the measurements in the dedicated environment.

The exact reasons are unclear, it may be that putting workers too often to sleep, leads to lots of context switches between the processes which worsens the throughput of the whole system.



(a) Graph Bipartitioning Benchmark



(b) LU decomposition benchmark

Figure 4.4: Graph Bipartitioning and LU factorization benchmarks in multiprogrammed environment on Mars that show two contrary performance characteristics regarding the usage of a backoff

However, it is very interesting that the results are again highly dependent on the type of tasks spawned by the application.

One reason for this behavior might be that the graph bipartitioning benchmark is computation bound, whereas the LU decomposition benchmark is memory bound. In the bipartitioning benchmark a graph with only 35 nodes is used, which allows the processors to keep the data in its caches most of the time. The LU decomposition benchmark operates on matrices of size $1024x1024$ which is a lot more of data.

4.4 New Backoff variants

Since Pheet allows to exchange the backoff using template parameters, it is very easy to implement different backoff variants and configure their parameters. In Pheet a backoff must implement only two functions:

`backoff()` This function is called, whenever the worker decides that it is beneficial to block the thread for some time, since it cannot find any useful work to process. E.g., this function is called after an unsuccessful steal attempt.

`reset()` The `reset` function is in a way the counterpart to the `backoff` function. It is called whenever the thread finds enough tasks to execute and the backoff time should be decreased or completely reset. One situation where this happens is when a task is successfully stolen from another worker.

Beside the already mentioned versions without backoff and the default Pheet exponential backoff, we decided to implement the following backoff variations:

Yield backoff This backoff mimics the behavior of the classic ABP work-stealing scheduler and just calls the `yield` function when the thread backs off.

Exponential backoff Pheet uses an exponential backoff at the moment, for better comparison we implemented another version of such backoff that allows a more fine-grained customization of its parameters.

Linear backoff It is interesting to compare the more complex backoffs to a very simplistic linear backoff.

Combined backoff The idea of the combined backoff is based on the exponential-linear backoff. In the exponential-linear backoff the backoff time is increased in an exponential way as long as there are only a few failed steal attempts and it is switched to increasing linearly above a threshold. However, we also want to try it the other way round, where the backoff time is increasing linearly first and exponentially later.

Linear combination backoff In this backoff implementation the backoff time is calculated by linear combination of two different backoffs. A parameter controls the weighting between the two backoffs. A more detailed description of the backoff is given further below.

Beside these basic backoff types we try to evaluate the influence of the different properties of the backoff. We expect that the minimum and maximum backoff time are the most crucial parameters. Additionally the way how the backoff time is decreased may affect the performance. We also test whether using some kind of randomness changes the behavior. Since each worker already uses randomness when selecting victims for stealing tasks, it is not expected that random sleep times in the backoff are beneficial.

A further idea that arose during the implementation of the backoffs, is a combination of `yield` with other backoff variants. The idea is that the first call of `backoff` executes a simple `yield` system call, whereas all next calls until `reset` behave like the default backoff implementation. So when `backoff` is only called once the worker is not actually put to sleep, but just the `yield` function is executed. All succeeding calls, until `reset` is called, behave like a normal backoff and put the thread to sleep.

In the next few pages we describe properties and selected default parameter values for the different kind of backoffs. The selection of default parameter values is mostly done by running a set of benchmarks. Since testing all these parameters takes a lot of time, the benchmark procedure is modified for this section: For the *Unbalanced Tree Search* benchmark only the variant with a maximum depth of $d = 10$ is used. In all benchmarks only 7 iterations instead of 20 are executed on Saturn.

Linear backoff

Beside the yield backoff that does not contain any configurable properties, the linear backoff is the simplest backoff. Nevertheless there are still many parameters that can be chosen:

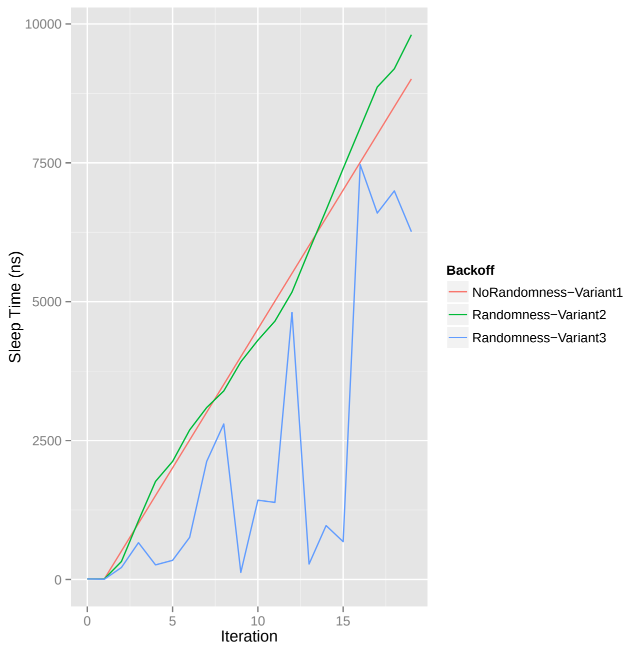


Figure 4.5: Exemplary sleep times of different randomness variants used in linear backoff

Step size The step size defines the constant value that is added to the backoff time at every `backoff` call.

Minimum and maximum backoff time Sensible values for the minimum and the maximum time must be chosen.

Slow reset We use the linear backoff to find out, if it makes sense to decrease the backoff time slowly when `reset` is called. When the slow reset mechanism is used, every time `reset` is called, the backoff time is decreased by the step size until the minimum backoff time is reached.

Randomness We test three variants of randomness. First of all we do not use random values at all. In this case every time the backoff is called, the time to sleep is simply increased by the step size. In the second variant, the backoff time is increased by $STEP\_SIZE/2 + Rand(0, STEP\_SIZE)$. In the third variant the backoff time is constantly increased by the step size. However, the thread is not actually suspended for this backoff time, but for a random value between 0 and the current backoff time.

Figure 4.5 shows exemplary backoff times of the three randomness variants of the linear backoff. One can see that first variant that does not use any random value and the second variant which increases the backoff time in a random way, behave very similarly. Contrary the third variant has very varying sleep times, since any random value between 0 and the current backoff time is possible.

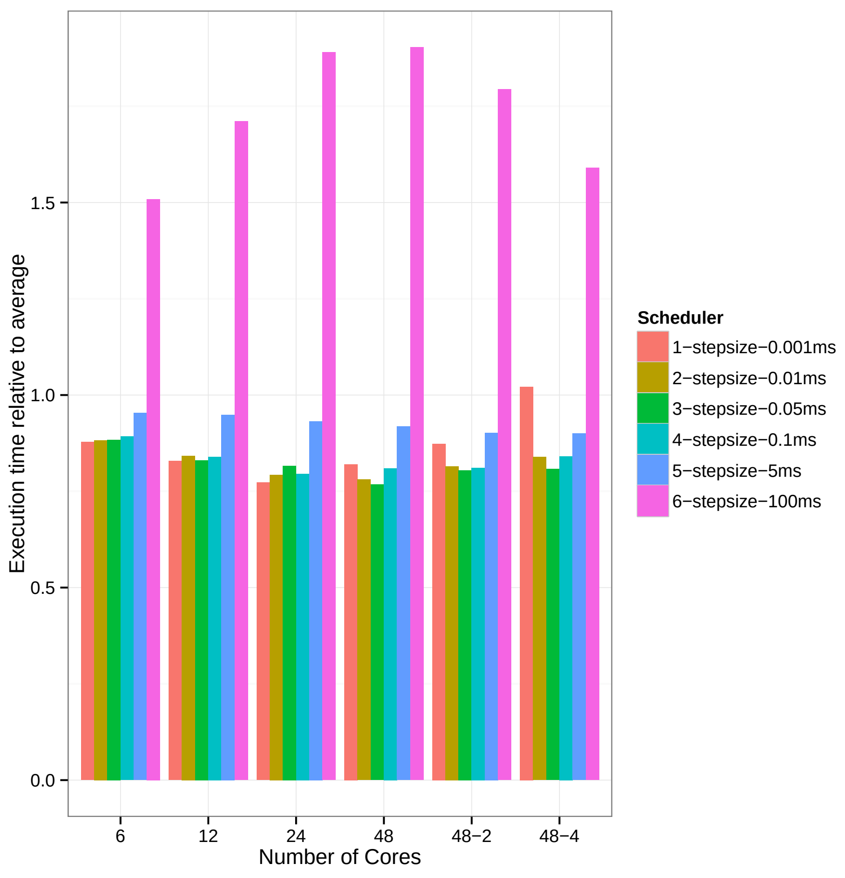


Figure 4.6: Aggregated results showing impact of different step sizes for linear backoff on Saturn

Step size

To determine which values for the step size are reasonable, the minimum backoff time was set to $100ns$ and the maximum backoff time to $1s$. $1s$ as maximum backoff time is a quite large value and is probably never reached in our benchmarks. We chose this large value, to make sure that the different step sizes have a large influence on the results. No randomization was used and the backoff time was set back to its minimum value at each `reset` call. This means *slow reset* was not used. The tests were executed on Saturn with step sizes between $1\mu s$ and $0.1s$. All benchmarks show similar behavior according to the step sizes. Figure 4.6 shows the aggregated results of all benchmarks. One can see that step sizes between $10\mu s$ and $0.1ms$ are reasonable for the linear backoff in these situations, also when multiple instances are run in parallel.

It seems that especially the benchmarks in the multiprogrammed environment are more sensible to the step size. A step size of $1\mu s$ is roughly as good as a step size of $50\mu s$, as long as only one instance is run at a time. When two and more particularly four instances are run in parallel, $1\mu s$ is clearly worse. Comparing all results we think $50\mu s$ is the best suited step size for our linear backoff. We use this step size in all further benchmarks of the linear backoff.

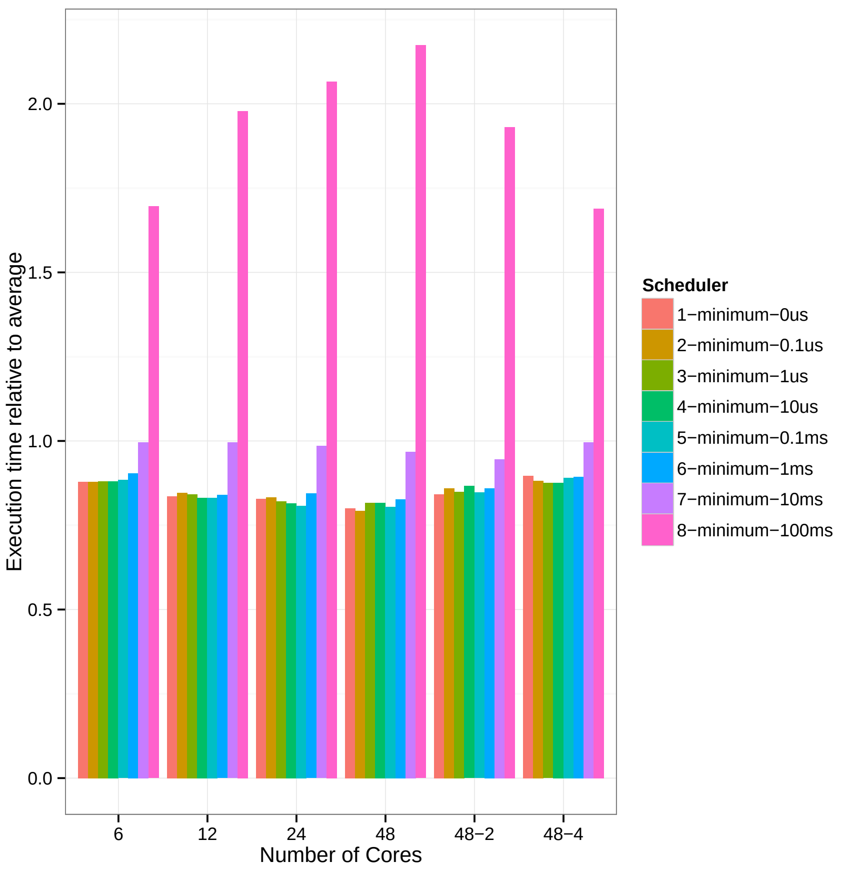


Figure 4.7: Aggregated results showing impact of different minimum backoff times for linear backoff on Saturn

Minimum backoff time

To find out the optimal minimum backoff time the maximum backoff time was again set to $1s$ and the step size was set to $50\mu s$. Minimum backoff times between $0s$ and $0.1s$ were tested. The results in Figure 4.7 show that the performance is quite similar as long as the minimum time is below $10ms$. It is interesting that there is hardly any difference between $0ms$ and $1ms$. Therefore $10\mu s$ was chosen as default minimum backoff time.

Maximum backoff time

In the benchmarks to determine the best maximum backoff time the same parameters like in the tests of the minimum backoff time were used, but the minimum backoff time was set to $10\mu s$. Maximum backoff times between $10\mu s$ and $1s$ were tested. In the case were the maximum backoff time was set to $10\mu s$, the thread is stopped for exactly $10\mu s$ every time, since this is also the minimum backoff time.

The tests showed the importance of setting a large enough maximum backoff time in mul-

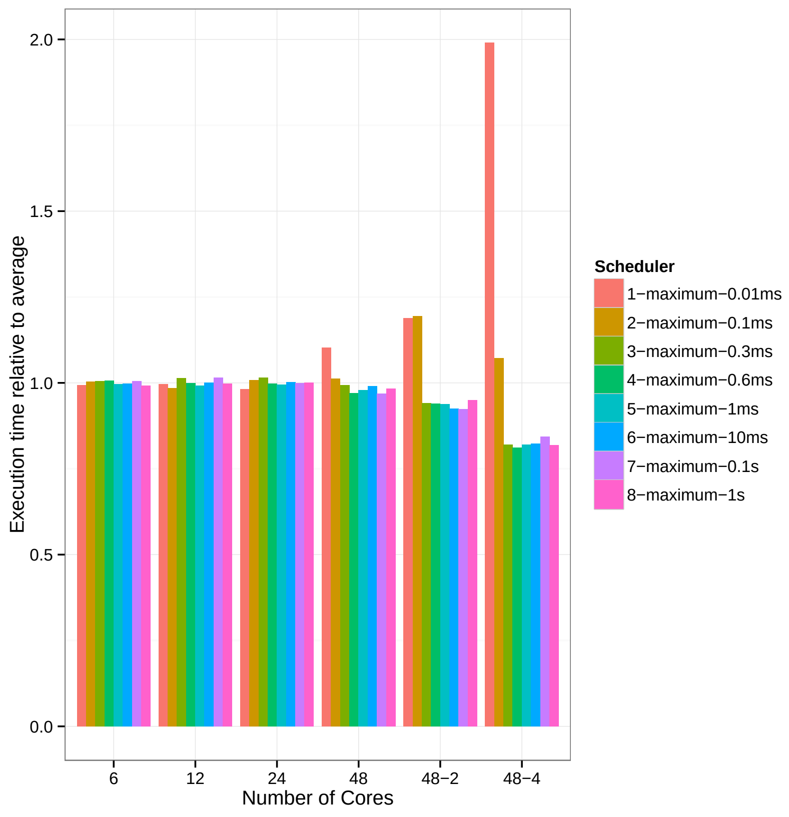


Figure 4.8: Aggregated results showing impact of different maximum backoff times for linear backoff on Saturn

tiprogrammed environments. Depending on the test case, all values above $10\mu s$ or $100\mu s$ lead to very similar performance results. It is no big surprise that it does not matter if the maximum backoff time is set to $1ms$, $10ms$ or even greater values. Since the step size is set to $50\mu s$, `backoff` would have to be called 20 times to reach a backoff time of $1ms$. It is likely that sometime before a successful steal occurs.

However, these results do not necessarily mean that the maximum backoff time can be any large enough value or that the maximum time is not important at all. It can happen that a worker does not find a task for a very long time and therefore steadily increases the backoff time. If there is no maximum value, the worker could be blocked for a very long time and e.g. block the other threads when all tasks are already finished. Therefore as default value we suggest a small one that performs well, in our test cases this would be something around $0.5ms$. $0.5ms$ will also be the default value for our linear backoff.

Randomness and slow reset

In a final step the parameters for adding some randomness and resetting the backoff time slowly, were tested. Figure 4.9 depicts the results. In the *no-rand* test case no randomness was used

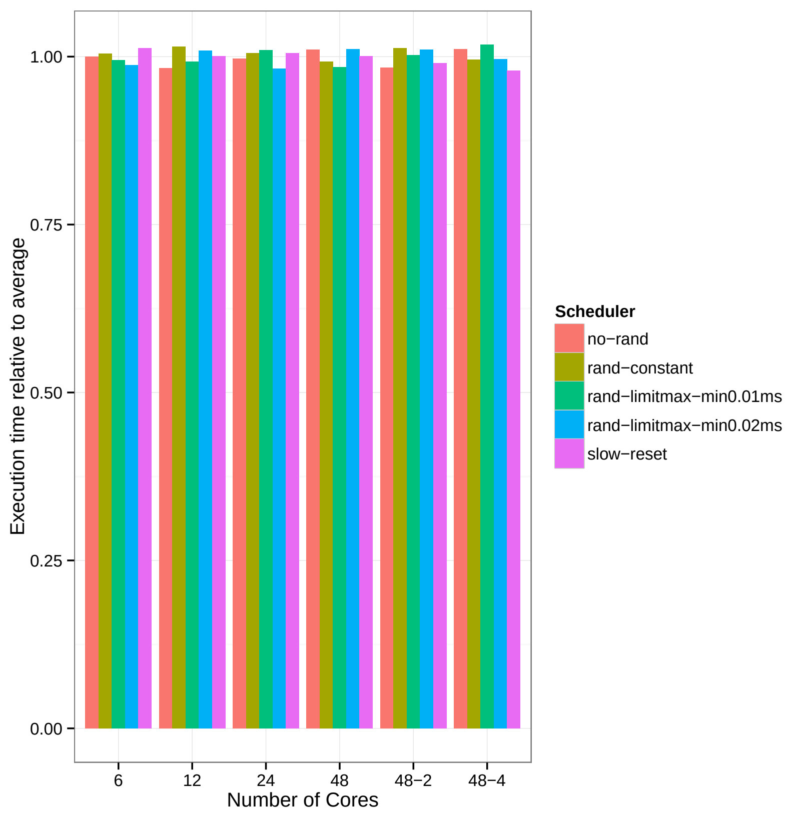


Figure 4.9: Aggregated results showing impact of different randomness variants and slowly resetting the backoff time in linear backoff on Saturn

and the backoff time was not reset slowly. This is the same behavior like in the tests that were presented so far. The *rand-constant* case shows the results, where the backoff time is increased by $STEP\_SIZE/2 + Rand(0, STEP\_SIZE)$ (variant 2). The described variant 3 is used in the *limit-max* test cases. The backoff time is some random value between 0 and the calculated backoff time. Since this leads to smaller backoff times in average, two scheduler variants were created: One variant uses $10\mu s$ as minimum backoff time and the other $20\mu s$. The last tested variant is the *slow-reset* case, where the backoff time is reset slowly. No randomness is used in this test case.

The results do not show any significant differences between all this described scheduler variants. This is especially interesting for the slow-reset case, we expected that this influences the performance notably.

Overview linear backoff results

In summary the results of the parameter tests for the linear backoff are insightful. We expected that the backoff is much more sensitive to changing the parameter values. As a matter of fact, this is not the case in our tests. It will be interesting to see if the other backoff variants show

Parameter	**reasonable value range**	**chosen default value**
step size	$10\mu s$ - $0.1ms$	$50\mu s$
minimum backoff time	$0ms$ - $1ms$	$10\mu s$
maximum backoff time	$> 100\mu s$	$0.5ms$
randomness	no significant influence	no randomness
slow reset	no significant influence	no slow reset

Table 4.1: Reasonable and chosen parameter values for linear backoff

similar behavior.

Table 4.1 shows an overview of reasonable value ranges and the values we chose as default parameter values for our linear backoff implementation. Of course like in many other components in Pheet, this default values can be overwritten by the developer if needed.

As a variation of the linear backoff, we tried to combine it with the `yield` operation: In this variation the first invocation of `backoff` calls yield instead of putting the worker to sleep. All other backoff invocations then follow the same semantics as the normal linear backoff. We could not measure any significant performance differences when using this variation and therefore did not follow the approach any further.

Exponential backoff

Most of the time exponential backoffs are implemented as binary exponential backoffs. We want to find out whether other exponents than 2 are usable for backoffs in this area. We are also interested if the usage of non-integer exponents is reasonable or even beneficial.

Backoff factor r

To determine the best backoff factor r we set the maximum backoff time to $1s$. Backoff factors between 1.1 and 3.5 are tested. We expected that the minimum backoff time also has a big influence on the performance. Therefore we executed all tests with a minimum backoff time of $10ns$ and $10\mu s$.

The benchmarks showed diverse results in these test runs. Some of them benefit from larger backoff factors, others run faster with lower factors. Figure 4.10 shows an overview of the results for the tests with a minimum backoff time of $10ns$. The test runs with a minimum backoff time of $10\mu s$ showed very similar characteristics.

In this case it is not easy to pick one backoff factor that performs the best. We decided to use 1.3 as our default backoff factor. This factor offers acceptable performance in all different benchmarks. However, it must be said that depending on the benchmark other benchmark factors perform better.

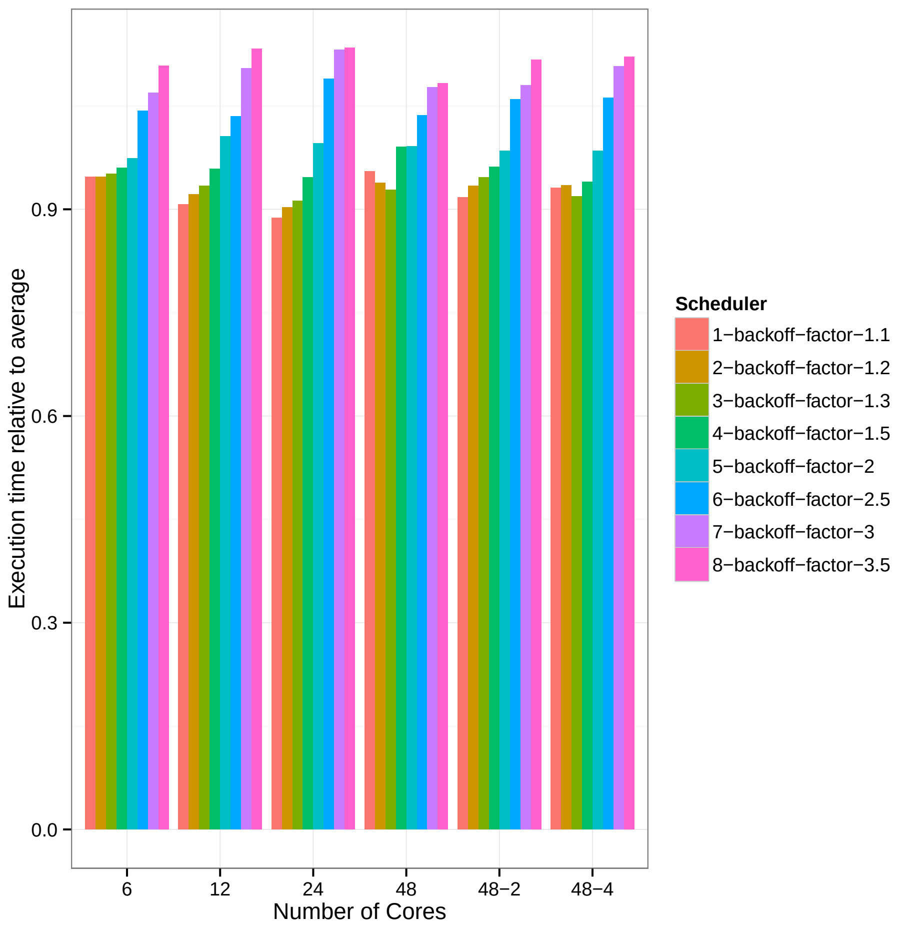


Figure 4.10: Aggregated results showing impact of different backoff factors in exponential backoff on Saturn

Minimum backoff time

To find out the best minimum backoff time, the maximum backoff time was set to $1s$ and the backoff factor to 1.3. Minimum backoff times in the range from $10ms$ to $0.1s$ were tested.

There were no big differences between all chosen minimum backoff times up to $10ms$, as can be seen in Figure 4.11. Like in the linear backoff, we chose $10\mu s$ as default value for the minimum backoff time of the exponential backoff.

Maximum backoff time

For the tests regarding the maximum backoff time, the same parameters that were used to determine the best minimum backoff time were used. The minimum backoff time itself was set to $10\mu s$. Maximum backoff times in the range from $0.1ms$ to $1s$ were tested.

Like in the case of the linear backoff, the maximum backoff time does not have a big in-

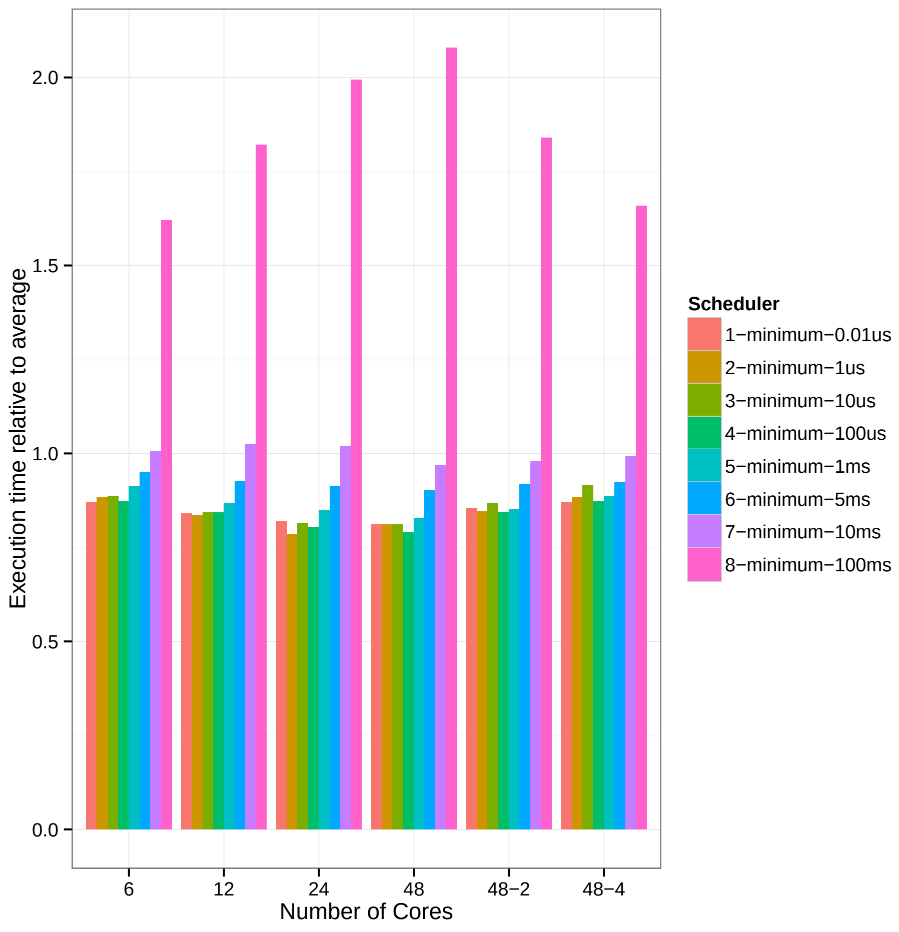


Figure 4.11: Aggregated results showing impact of different minimum backoff times in exponential backoff on Saturn

fluence on the overall performance. Only in multiprogrammed environments maximum backoff times of $0.1s$ or larger are worse than smaller ones. Between the other tested variants basically no differences exist, as can be seen in Figure 4.12. We therefore chose $0.5ms$ as default maximum backoff time.

Randomness and slow reset

In our implementation of the exponential backoff two different kinds of randomness can be used:

Const In this variant, the new backoff time is calculated by adding some random value in the range from $-MinBackoff/2$ and $MinBackoff/2$ to the calculated backoff time: $OldBackoffTime * r + Rand(-MinBackoff/2, MinBackoff/2)$

LimitMax In the second variant the backoff time is calculated in the same way like in the *Const* variant. However, the thread is not blocked for the calculated backoff time, but

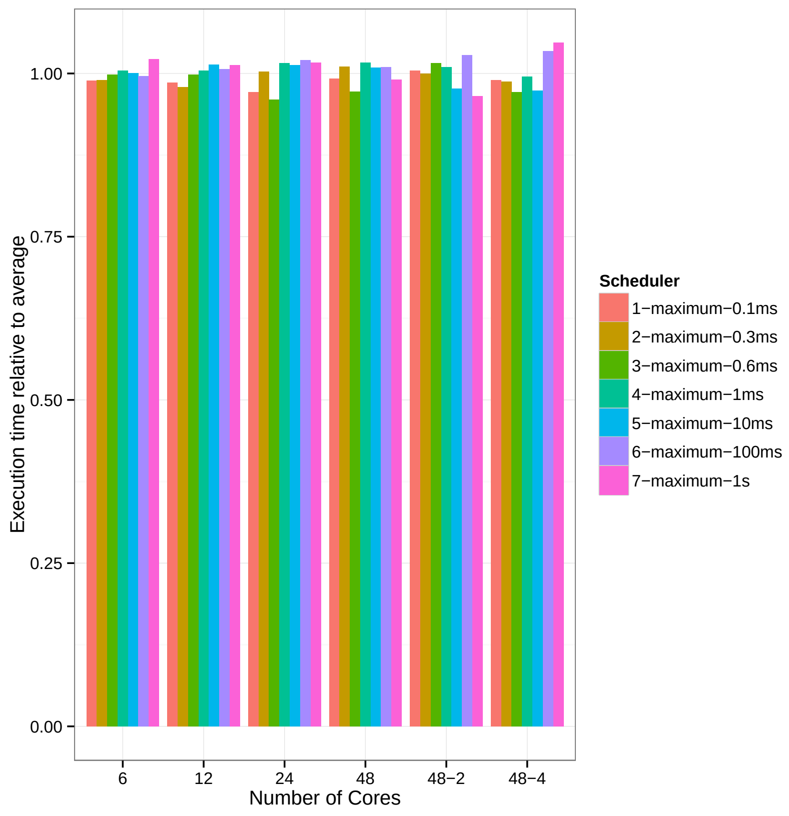


Figure 4.12: Aggregated results showing impact of different maximum backoff times in exponential backoff on Saturn

for a random timespan between 0 and the calculated backoff time. Since in this case the expected time to sleep is lower, tests with a minimum backoff time of $20\mu s$ instead of $10\mu s$ were made.

Figure 4.13 shows a comparison between the two variants where some randomization is included in the backoff time, one variant were *slow reset* is used and a variant without randomization or slow reset. Like in the linear backoff, no significant differences between all these variants can be seen. Neither randomization nor slowly resetting backoff time improves the performance of the scheduler.

Overview exponential backoff results

Like in the case of the linear backoff, choosing good parameter values for the exponential backoff is not as hard as we expected. As long as the parameter values are within a reasonable range of values, the performance of the scheduler is not influenced a lot.

Table 4.2 shows an overview of reasonable value ranges and the values we chose as default parameters for our exponential backoff implementation. Again this default values can be overwritten by the developer if needed.

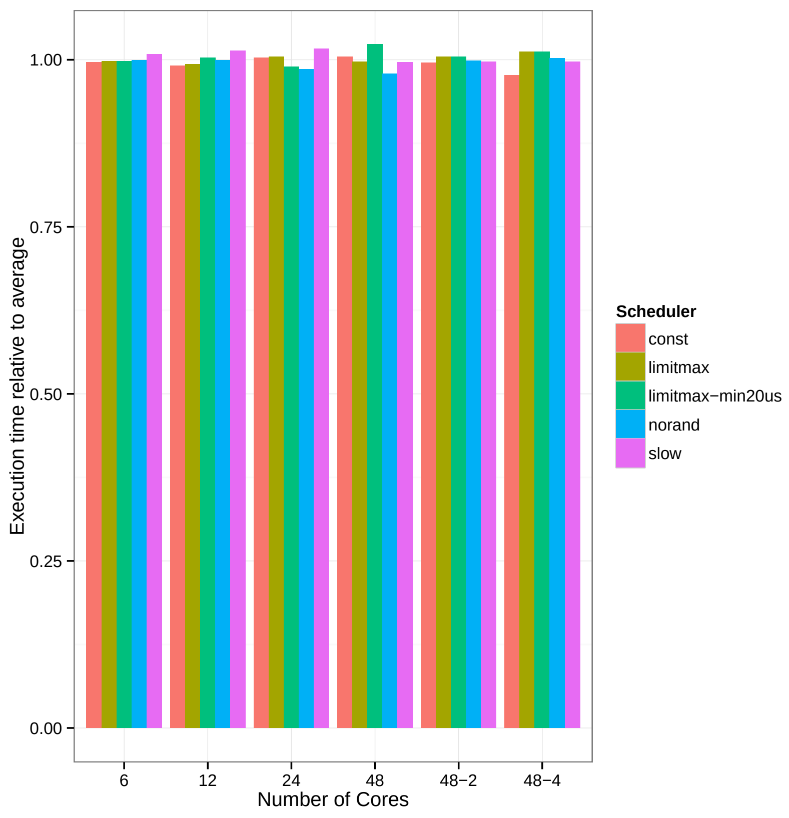


Figure 4.13: Aggregated results showing impact of different randomization variants and slow reset test in exponential backoff on Saturn

Parameter	**reasonable value range**	**chosen default value**
backoff factor	$1.1s$ - $2s$	$1.3s$
minimum backoff time	$10ns$ - $1ms$	$10\mu s$
maximum backoff time	$10ns$ - $10ms$	$0.5ms$
randomness	no significant influence	no randomness
slow reset	no significant influence	no slow reset

Table 4.2: Reasonable and chosen parameter values for exponential backoff

Combined backoff

The idea of the *combined backoff* is based on the exponential-linear backoff. In the exponential-linear backoff the backoff time is increased in an exponential way until some threshold and afterwards in a linear way [28]. We want to test this *exponential-linear backoff* and a second variant where the backoff time is first increased linearly and then exponentially. This second variant is called *linear-exponential backoff*.

Listing 4.1: CombinedBackoff Template

```cpp
template <class BackoffOne, class BackoffTwo>
class CombinedBackoffImpl {
public:
  void backoff();
  void reset();
  unsigned int current_sleep_time() const;

  static_assert(BackoffOne::VALUE_MAX_BACKOFF <= BackoffTwo::
      VALUE_MIN_BACKOFF,
      "Max BackoffTime of BackoffOne must be smaller or equal to Min
          BackoffTime of BackoffTwo");

  constexpr static unsigned int VALUE_MIN_BACKOFF = BackoffOne::
      VALUE_MIN_BACKOFF;
  constexpr static unsigned int VALUE_MAX_BACKOFF = BackoffTwo::
      VALUE_MAX_BACKOFF;
  constexpr static bool VALUE_SLOW_RESET = BackoffOne::VALUE_SLOW_RESET ||
       BackoffTwo::VALUE_SLOW_RESET;

private:
  BackoffOne bo1;
  BackoffTwo bo2;
};

template <class BackoffOne, class BackoffTwo>
void CombinedBackoffImpl<BackoffOne, BackoffTwo>::backoff() {
  if (bo1.current_sleep_time() >= BackoffOne::VALUE_MAX_BACKOFF) {
    bo2.backoff();
  }
  else {
    bo1.backoff();
  }
}

// other methods

// using directives that compose actual usable backoffs
template <class Pheet>
using LinearBackoffExpLin = LinearBackoffImpl<Pheet, 10, 3500, 500,
    BackoffRandomness::Without, false>;
template <class Pheet>
using ExponentialBackoff2ExpLin = ExponentialBackoff2Impl<Pheet, 3500,
    100000, 2, BackoffRandomness::Without, false>;
```

```
template <class Pheet>
using ExponentialLinearBackoff = CombinedBackoffImpl<LinearBackoffExpLin<
    Pheet>, ExponentialBackoff2ExpLin<Pheet>>;

template <class Pheet>
using YieldExponentialLinearBackoff = CombinedBackoffImpl<YieldBackoff<
    Pheet>, ExponentialLinearBackoff<Pheet>>;
```

Listing 4.1 shows the implementation of this combined backoff. `CombinedBackoff` is a template for backoffs that can be composed by combination of two backoffs. It is used for the implementation of the exponential-linear backoff, the linear-exponential backoff and the exponential-linear backoff with yield. The `ExponentialLinearBackoff` consists of a linear and an exponential backoff. The `YieldExponentialLinearBackoff` is composed by combining the `YieldBackoff` and the `ExponentialLinearBackoff`. The composition of these backoffs is achieved by the using directives at the end of the listing. Since `CombinedBackoff` is a template, this compositions already exist at compile time and other combined backoffs can be created by the user of the Pheet library. This shows a bit of the flexibility of Pheet and the C++ template mechanism.

Since two backoffs are involved, the number of parameters that can be chosen is even larger. Due to time and resource constraints we cannot test all reasonable combinations of parameter values. We test a set of value ranges for this parameters which we think are most promising. However, based on the results of the linear and exponential backoff tests, we expect that the parameter values do not matter that much in this case either.

To decide whether it is plausible that tuning the parameters improves performance, we test three simple variations of both the linear-exponential and the exponential-linear backoff: We check if significantly changing the threshold between the linear and the exponential backoff influences the performance noteworthy. The parameter values used in these three variants are shown in Table 4.3. Minimum backoff time *1* and maximum backoff time *1* denote the times for the exponential backoff and maximum backoff time *2* for the linear backoff in the exponential-linear backoff. For the linear-exponential backoff it is the other way round. In other words, maximum backoff time 1 is the threshold where it is switched between the two backoffs. This threshold is adjusted in the three variants: In variant *1* it is roughly in the middle of the two backoffs, in variant *2* the second backoff is used only when `backoff` is called very often, in third variant the threshold is very low so the second backoff is used frequently.

We only show the results of the exponential-linear backoff in Figure 4.14. The linear-exponential backoff behaves very similarly. One can see that there are no significant differences between all these variants. We therefore decided that it does not make any sense to test other parameter values. It can be assumed that as long as the parameters are in a reasonable value range, the performance does not change a lot. We tested and described these reasonable ranges in-depth for the exponential and the linear backoff.

As default parameter values for our implementation of the exponential-linear and the linear-exponential backoffs we chose the ones from variant *1*, as given in Table 4.3.

Parameter	**Variant**	**Exponential-linear**	**Linear-exponential**
backoff factor	all variants	2	1.3
backoff step	all variants	$50\mu s$	$25\mu s$
minimum backoff time 1	all variants	$10\mu s$	$10\mu s$
maximum backoff time 1 minimum backoff time 2	variant 1	$200\mu s$	$100\mu s$
	variant 2	$400\mu s$	$400\mu s$
	variant 3	$50\mu s$s	$50\mu s$
maximum backoff time 2	all variants	$0.5ms$	$0.5ms$

Table 4.3: Parameter values for three different exponential-linear and linear-exponential backoff variants

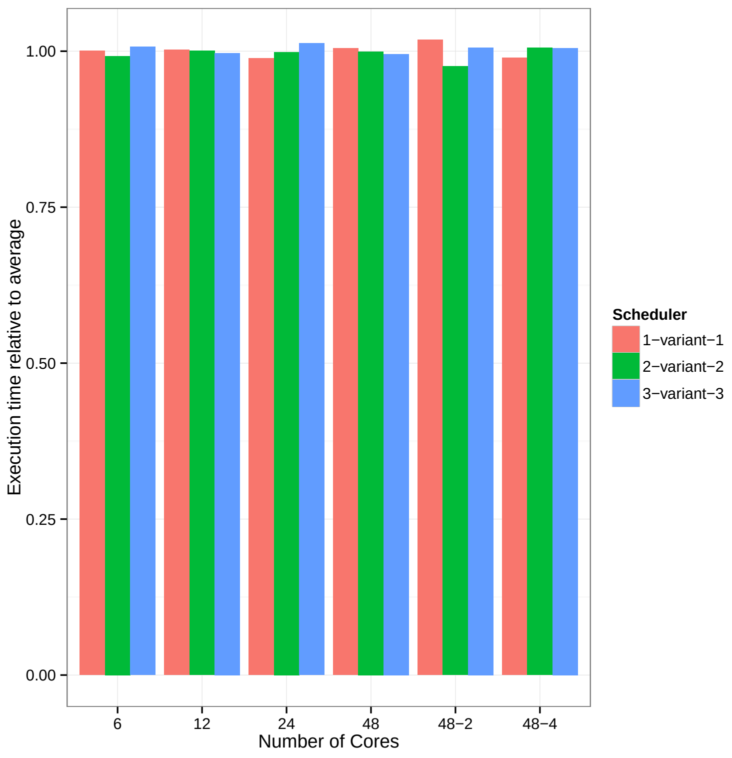


Figure 4.14: Aggregated results showing impact of different variants of exponential-linear back-off on Saturn

Linear combination backoff

As a variation of the combined backoff we implemented a *linear combination backoff*, where the backoff time is calculated as a linear combination of a linear and an exponential backoff. The backoff time is therefore calculated in the following way:

$$bt_c = t * tb_1 + (1 - t) * tb_2 \tag{4.1}$$

In Equation 4.1 tb_c denotes the combined backoff time, tb_1 and tb_2 denotes the backoff times of the two used backoffs and t is a parameter used to control the linear combination of the values, were $0 \leq t \leq 1$.

Our implementation of the linear combination backoff can use any kind of backoff, it is not limited to an exponential and a linear backoff. However, we did not perform any tests using other backoffs.

Like in the case of the combined backoff, it is very time consuming to test a range of parameter values in a systematic way. So again we try to decide whether an in-depth analysis makes sense first. We test a few variants which should show significant performance differences if the backoff is sensible to changing the parameters. Based on the experiences with the other backoffs, we assume that again it is only important to choose reasonable parameter values. We expect that the value of t does not influence the results significantly, as long as the parameters of both used backoffs are chosen reasonable. Of course t would influence the performance, if the parameters of one of the two backoffs are set to completely senseless values.

We therefore executed the benchmarks using a linear combination backoff with t set to 0.1, 0.25, 0.5, 0.75 and 0.9. The higher t is, the higher is the influence of the linear backoff. The parameter values for the linear and exponential backoff are set to the chosen default values that can be seen in Table 4.1 and Table 4.2 respectively.

The results are again very similar for all different t values. However, in multiprogrammed environments the variants with a higher t value perform slightly better in some benchmarks. Figure 4.15 shows exemplary results for the sorting benchmark.

Nevertheless we think that the potential for optimization of the parameters is not very high. Therefore we did not execute further tests with different parameters and chose $t = 0.75$ as our default parameter for the linear combination backoff that uses the default linear backoff and exponential backoff that were already presented.

Parameter tuning conclusion

The parameter tuning of the different backoff implementations showed that the overall performance does not change a lot, as long as the parameter values are in a reasonable range. In the end optimal parameter values highly depend on the machine where the application is executed and on the tasks that are executed. In the parameter tuning tests we often saw the effect that different benchmarks would require different parameters. Due to time and resource constraints we could not execute all tests also on a second machine, but we also ran some of the tests on Mars. These tests showed that sometimes slightly other parameter values would be better on this machine. We therefore doubt that it is possible to select one ”perfect set” of parameter values.

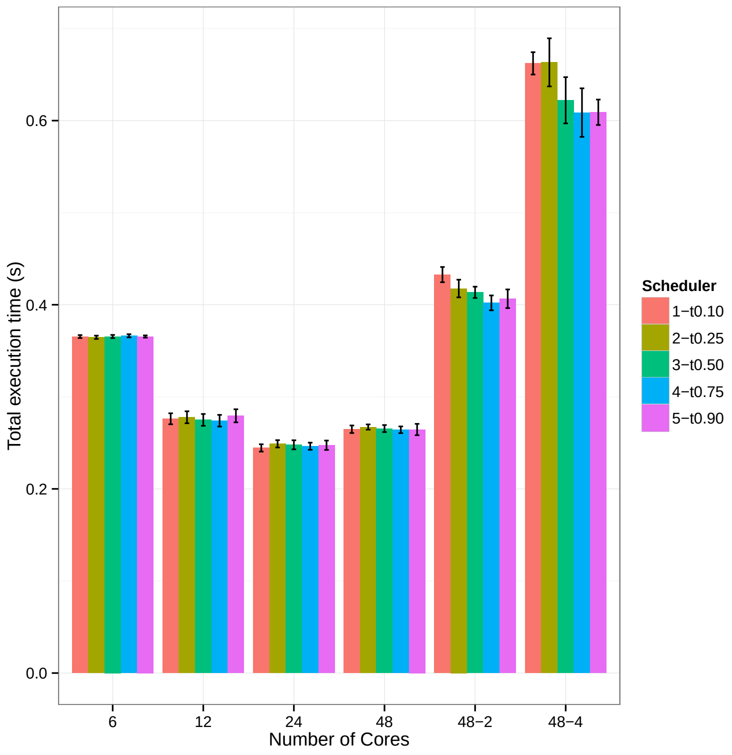


Figure 4.15: Results of quicksort benchmark showing different variants of linear combination backoff on Saturn

Figure 4.16 shows the sleep times for the different backoff implementations, using the described default parameter values. The x-axis shows the number of times `backoff` was called without calling `reset` and the y-axis shows the corresponding sleep time in nanoseconds. One can see that most of the new backoffs show a very similar behavior. One slight exception is the exponential backoff that increases the backoff time significantly slower than the other variants. The reason for this is the small chosen backoff factor of 1.3. The backoff that is currently used in Pheet (*Original*) increases the backoff time even slower than our new exponential backoff. It has both a smaller minimum and maximum backoff time, additionally it uses randomization by default. It will be interesting to see how these different backoff implementations perform in comparison.

4.5 Comparison of backoffs in benchmarks

To compare the different backoff types the described benchmarks were run on Saturn and Mars. 20 iterations of each benchmark were run, the graph bipartitioning benchmark was executed 100 times. The benchmark runs show that all tested backoffs have similar performance characteristics as long as reasonable parameters are used. It does not make a big difference which backoff type is used, as long as its parameters are selected with care. A nice result is that our carefully

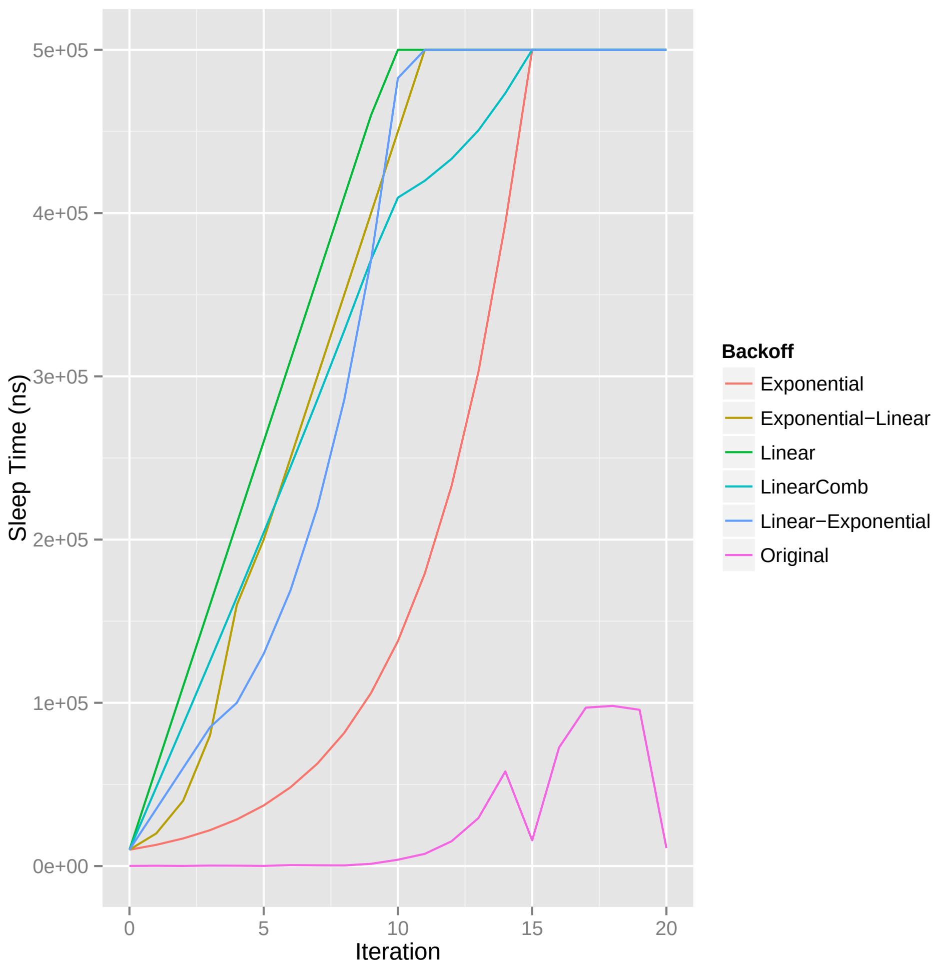


Figure 4.16: Backoff times of different backoffs

tuned backoff types overall perform better than the backoff used so far in Pheet. Additionally the schedulers using our new backoff types perform better than the scheduler that uses a simple `yield` call.

As a conclusion of this chapter we want to present a few interesting and surprising results of these tests.

Figure 4.17 shows the results of the LU decomposition benchmark on Mars. The results show that like in most of the other tests our new backoff variants perform very similar. One can see that the original backoff and the yield backoff perform very badly when 80 or more cores are used. An explanation for this could be that the yield backoff does not increase the backoff time at all and the original backoff increases the backoff time very slowly. Therefore one could argue that workers waste too much time trying to steal tasks when there is not enough work available. Compared to these two backoffs, the variant without any backoff (*noop*) performs pretty well.

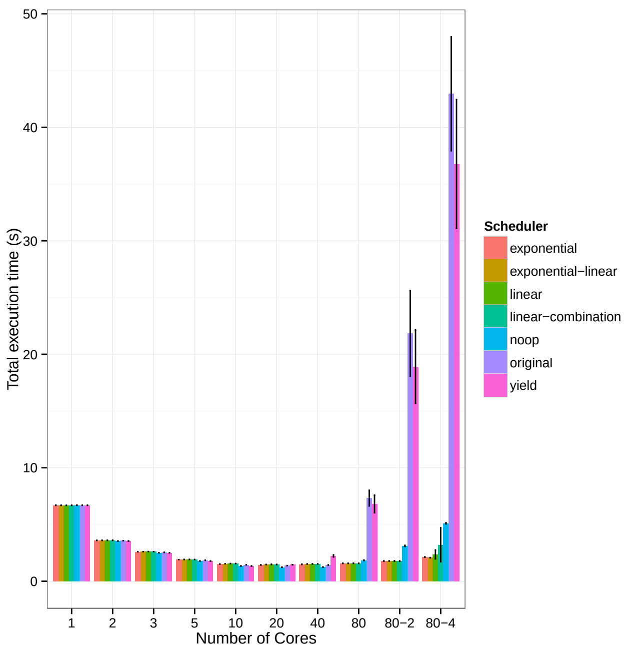


Figure 4.17: Test results of LU decomposition benchmark run on Mars

However, one would think that the variant without any backoff would waste at least as many cycles with unsuccessful steal attempts as the variant with `yield`.

On Saturn in the same benchmark the yield backoff performs roughly as good as our new backoff variants, which can be seen in Figure 4.18. In the multiprogrammed environment tests on Saturn, the variant without any backoff and the original backoff are worse than the other variants.

Since we were surprised of this behavior, we rerun the LU decomposition benchmark on both machines and got the same results again. One explanation for this behavior on Mars may be that in this benchmark it is bad to put a thread to sleep for a short time or to call `yield` very often. It seems to be better not to do anything (*noop*) and just let the operating system decide when a thread should be suspended.

Figure 4.19 shows an overview of the performance of the different backoffs on Saturn, Figure 4.20 contains such overview for the tests on Mars.

On both machines the variant using no backoff at all and Pheet's original backoff implementation are significantly slower than the other variants. This is especially true, when multiple tests were run in parallel. It is interesting to see that the `yield` variant works quite well on Saturn, whereas it performs bad on Mars.

The benchmarks also showed that despite the optimization efforts, in a few situations it is still beneficial to completely disable the backoff. Therefore further algorithms are also compared with the scheduler that does not use any kind of backoff.

However, overall the new backoff variants work quite well compared to the backoff used so far in Pheet and also compared with the variant that does not use a backoff at all. Our new

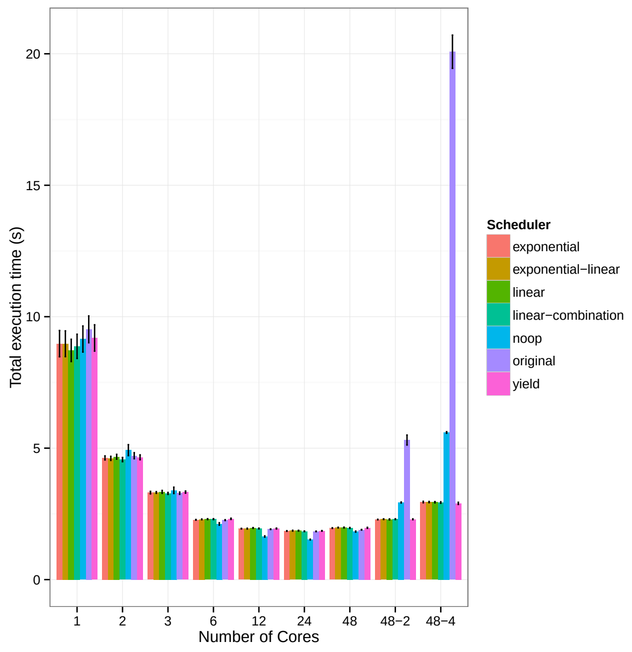


Figure 4.18: Test results of LU decomposition benchmark run on Saturn

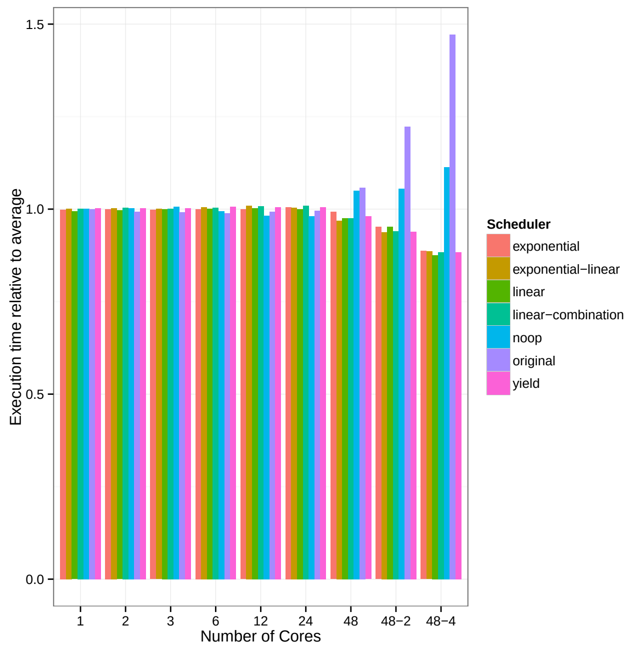


Figure 4.19: Aggregated results of backoff benchmarks on Saturn

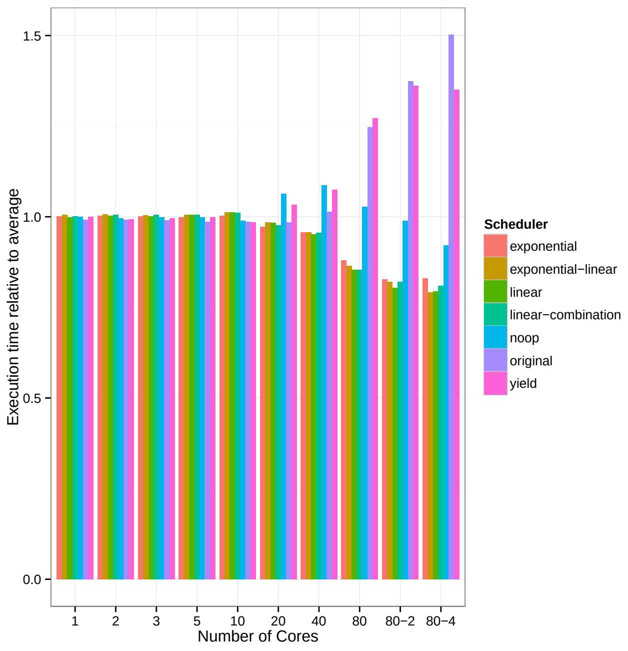


Figure 4.20: Aggregated results of backoff benchmarks on Mars

backoffs additionally do not have these bad worst case scenarios that were shown in the section about the analysis of the existing scheduler. Therefore we think that one of the new backoffs can replace the existing backoff in Pheet as a new default backoff. In our tests the backoffs perform very similar, so there is no clear best one. Since the linear backoff is probably the simplest one, we chose the scheduler using the linear backoff as our new baseline for further comparisons.

CHAPTER 5

Adaptive Work-Stealing Algorithms

Most of the presented approaches for adaptive work-stealing use some kind of central entity to decide how many threads should be used to process the tasks. This can lead to scalability issues when many processing cores are available. Balanced work-stealing that was introduced in Section 2.6, does not have a central entity, but it relies on special operating system features. As long as the required features are not available on a majority of the used operating systems, this is not a very practical technique.

Therefore, in this section a technique for adaptive work-stealing with the following objectives is presented:

No central data structure The usage of a central data structure can lead to high contention when all cores try to access the data at the same time. The more processing cores are used, the worse this issue gets.

No central process or service A central process can also lead to scalability problems and somehow it must be controlled that such process is started exactly once when an application that requires this process is run.

Portable The scheduler must neither be bound to a single operating system or make use of a special operating system feature, nor must it be bound to a special hardware architecture.

No fixed quantum length If there are quanta of fixed length, every thread periodically has to stop working to check if a given condition is met. This is an unnecessary overhead, since workers that have enough work to process should just be able to run without interruptions.

Configurable In accordance to most other aspects of Pheet, this extension should be also be as configurable as possible by the programmer.

We build upon the insights gained in the chapter about backoffs to design the adaptive work-stealing algorithm. Especially the gathered data about the existing scheduler and the tests regarding reasonable parameter values for the backoffs should be helpful. In this chapter we first describe our new algorithm, then give an overview of the performance of it and describe the differences to the scheduler that uses a simple backoff. We also present some implementation variants of the basic algorithm.

5.1 Introduction of dynamic quanta algorithm

We now introduce our adaptive work-stealing algorithm, which we call *dynamic quanta algorithm*. The existing scheduler we selected to adapt has the following basic properties, which we do not change for our extensions:

- For each available processor core a worker (thread) is created. These worker threads are created for each Pheet application that is run on the machine. Workers are not shared between multiple applications. Additionally the programmer can choose whether simultaneous multi-threading should be used or not.
- As victim selection policy the semi-random *hierarchical victim selection* is used.
- When a worker runs out of work, it tries to steal *one* task at a time from another worker. In the next steal operation it is first tried to steal a task from the same victim again.

Whenever it is decided that a worker is not needed for processing the tasks at the moment, the worker is put to sleep for some dynamically calculated timespan. This can be done in a portable way using the C++11 function `std::this_thread::sleep_for` or `sleep_until`. We thought about using an approach similar to the BWS algorithm, where workers are woken up by other workers when there are enough tasks available. In theory this seems like a superior approach, since workers can really be stopped as long as they are not needed. However, we could not think of a scalable solution and did not want to use mechanisms that are not part of standard operating systems. Additionally we think that it is better to waste a few more cycles with unsuccessful steal attempts than letting a worker sleep too long because it was not woken up by another worker earlier. This can happen easily when a BWS-like approach is used. Therefore we prefer the solution where a sleeping worker wakes up on its own.

The basic idea of most of the existing adaptive schedulers is to evaluate the workload of a worker based on the effort spent stealing and working. For that the execution of the application is partitioned into quanta and between two quanta the conditions for adding or removing workers are checked. As outlined the new approach should not interrupt workers as long as they are executing tasks and do not need to steal from other workers. Therefore an alternative for defining a fixed quantum length and pausing the workers at such intervals is needed.

In the Palirria algorithm a separate thread is used to check the conditions for adding or removing workers. However, we do not want to waste a thread and with it resources for this process. Our chosen alternative are *dynamic quanta*. The normal execution of the workers is only interrupted when it is decided that an interruption does not incur unnecessary overhead. A natural point where checking of thresholds should not delay the processing of tasks is when a worker tries to steal tasks from other threads. Only after an unsuccessful steal attempt it is checked if the current quantum has ended and whether the conditions to put a worker to sleep are met. There does not exist a fixed quantum length, but a minimum quantum length that is dynamically adjusted on the basis of the workload of the worker. A worker which task queue never gets empty, does not have any additional overhead because of the adaptive scheduling.

In the algorithm a quantum is classified as efficient or inefficient. This classification is based on the comparison of how much time was spent stealing with the time spent working in the previous quantum. When the previous quantum is classified as inefficient, the worker is put to sleep. The duration how long the worker is stopped, is again based on the time spent stealing. Additionally the number of consecutive inefficient quanta is taken into consideration.

At the end of every quantum the duration of the next quantum is calculated. The quantum length is dynamically adjusted by taking the number of consecutive inefficient quanta into consideration.

5.2 Dynamic quanta algorithm overview

In the description of the algorithm a set of variables is used. The notation of the variables follows three naming rules:

Variables which names start with n are integer values that are used as counters.

All variables that are named t denote point of times.

When a variable name starts with d, the variable stores some duration value.

Whenever a failed steal attempt occurs, a counter that stores the number of unsuccessful steal attempts since the last successful steal is increased. Additionally it is checked if the current quantum has ended. At the beginning of a quantum the time at which the quantum ends is set. Therefore the test if the quantum has ended can be simply done by comparing the stored quantum end time with the current time.

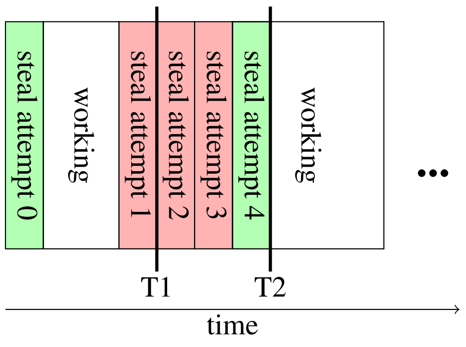


Figure 5.1: Illustration of the concept of a *first unsuccessful steal attempt* (T1)

To measure the time spent stealing we store the time when a *first unsuccessful steal attempt* occurs. We define a first unsuccessful steal attempt, as the first steal attempt of a series of unsuccessful ones. Therefore a first unsuccessful steal attempt is either the very first unsuccessful steal attempt of the worker or the previous steal attempt was a successful one. Whenever a successful steal occurs, the current time minus the time stored at the first unsuccessful steal attempt defines the duration of this series of unsuccessful steal attempts. Since we record the time *after* the first unsuccessful steal attempt, the time that the first steal attempt lasted, is not included. The reason for this approximation is that otherwise before every steal attempt the current time has to be recorded, which we want to avoid. We think this is a reasonable approximation, since a single steal attempt does not take a very long time.

Figure 5.1 illustrates this concept: *Steal attempt 1* is a *first unsuccessful steal attempt*, since the previous *steal attempt 0* is a successful one. Therefore the time *T1*, which is the time right after the unsuccessful steal attempt, is stored. After the next successful steal (*steal attempt 4*) the duration of this series of unsuccessful steal attempts can be calculated by $T2 - T1$.

The procedures for the failed and successful steals can be seen in Algorithm 5.1 and 5.2 respectively. The algorithms use the following set of variables that are all stored as instance variables of the worker:

quantum end time t_{qe} A time point that defines until when the current quantum at minimum lasts.

unsuccessful steals start time t_{us} Time point at which the last unsuccessful steal attempt occurred that until now was not followed by a successful one.

quantum steal time d_s Duration how long so far all steal attempts in this quantum lasted.

number of unsuccessful steal attempts n_{ua} In this variable the number of unsuccessful steal attempts since the last successful steal is stored.

Additionally the algorithms use the following function:

Now This is the function that returns the current time. In formulas the current time is denoted as n.

Algorithm 5.1: on-failed-steal

```
if $t_{qe}$ < Now then
    $d_s \leftarrow d_s + \text{Now} - t_{us}$;                update time spent stealing
    on-quantum-end();
else
    if $n_{ua} = 0$ then
        $t_{us} \leftarrow \text{Now}$ ;               first unsuccessful steal attempt
    end
end
$n_{ua} \leftarrow n_{ua} + 1$;     update number of unsuccessful steal attempts
```

The `on-failed-steal` method is the only method where it is checked, whether a quantum has ended. This means that a quantum can only end after a failed steal. Therefore a worker

Algorithm 5.2: on-successful-steal

```
1 if $n_{ua}$ > 0 then
2     $d_s \leftarrow d_s + \texttt{Now} - t_{us}$;                update time spent stealing
3     $n_{ua} \leftarrow 0$;
4 end
```

can only be put to sleep at this point. If there is no failed steal attempt at all, the very first quantum never ends and there is hardly any overhead for the adaptive extension of the scheduler. In such cases the only additional work is checking the condition in the `on-successful-steal` procedure.

The procedure that is invoked at the end of a quantum is shown in Algorithm 5.3. At the end of a quantum it is decided if the quantum was efficient or inefficient. If it was inefficient, the worker is put to sleep for some time. In both cases of an inefficient and an efficient quantum, some state variables are reset and a new quantum is started. This means the variable that stores the duration of the steal attempts in a quantum is reset and the quantum start time is set to the current time. Additionally the variable that stores the time of the first unsuccessful steal attempt is also set to the current time. A new quantum always starts after an unsuccessful steal attempt, therefore we know that at the beginning of the quantum the worker will try to steal another task. The function uses the following variables that were not described so far:

number of inefficient quanta n_{iq} This variable contains the number of inefficient quanta since the last efficient quantum.

quantum start time t_{qs} This time point contains the start time of the current quantum.

5.3 Detailed algorithm

In the previous section a quick overview of the algorithm was given. This section gives a more detailed description and discusses challenges that came up during the design of the algorithm.

As described there exists no fixed quantum length, but only a *minimum quantum length*. The programmer can set a default minimum quantum length d_{ql}. This default value is used as length of the first quantum and as length for all quanta that occur after efficient quanta. The minimum quantum length after an inefficient quantum is dynamically adjusted, but the default quantum length is used as base for the calculation of the length of such quanta. The exact calculation of this length is described further below.

Failed steal attempt

Every time a failed steal attempt occurs, the following two conditions are checked:

Check if last steal attempt was successful When the last steal attempt was successful, this means that this was a first unsuccessful attempt. Therefore the *unsuccessful steals start time* t_{us} is set to the current time.

Algorithm 5.3: on-quantum-end

```
1  $d_q \leftarrow$ Now $- t_{qs}$;            calculates actual length of quantum
2  $r_s \leftarrow d_s/d_q$;           calculates ratio of time spent stealing
3  if $r_s > T$ then
       quantum is classified as inefficient
4      $st \leftarrow$...;                          calculate time to sleep
5      Sleep($st$);                          block execution of thread
6      $n_{iq} \leftarrow n_{iq} + 1$;
7      $t_{qe} \leftarrow$...;                 calculate end time of next quantum
8  else
       quantum is classified as efficient
9      $n_{iq} \leftarrow 0$;
10     $t_{qe} \leftarrow$...; calculate end time of next quantum based on default
       quantum length
11 end
12 $d_s \leftarrow 0$;
13 $t_{qs} \leftarrow$ Now;
14 $t_{us} \leftarrow$ Now;
```

Check if current quantum has ended A quantum has ended, if the current time is greater than the *quantum end time* t_{qe}. If so, the procedure for the end of a quantum is invoked. Beforehand the *quantum steal time* d_s is increased by the duration between *now* and the *unsuccessful steals start time* t_{us}.

Successful steal

When a successful steal occurs, it is checked whether this steal is the first successful steal after one or more unsuccessful attempts. This can be checked using the variable that stores the number of unsuccessful steal attempts since the last successful one. When this variable is greater than zero, the quantum steal time d_s is increased and the number of unsuccessful attempts is reset to 0.

End of quantum

When the end of a quantum is reached, first of all the ratio between the time spent stealing and working in the previous quantum is calculated. Since the time spent stealing d_s and the length of the quantum $n - t_{qs}$ are known, this ratio can be calculated by:

$$r_s = \frac{d_s}{n - t_{qs}} \tag{5.1}$$

If r_s is greater than a threshold T the quantum is categorized as inefficient otherwise it is an efficient quantum. T is a parameter in the range $0 \leq T \leq 1$ that can be set by the developer who uses the scheduler.

When the quantum was *efficient* the worker continues execution without an interruption. The end time for the next quantum t_{qe} is set to $n + d_{ql}$ and the number of inefficient quanta n_{iq} is reset to 0.

The handling of inefficient quanta is the core of the algorithm. The basic idea is that the occurrence of an inefficient quantum means it is very likely that there are not enough tasks for all workers. Therefore the current worker is put to sleep for a certain amount of time. It remains to decide how long the sleep period of the worker should last. For this the following data is taken into consideration:

Ratio between time stealing and working r_s The higher this ratio is, the longer the worker should sleep.

Number of inefficient quanta n_{iq} The more consecutive inefficient quanta, the longer the worker should sleep.

Default quantum length d_{ql} The basis for the time to sleep is the default minimum length of a quantum.

The final equation to calculate the time of sleep d_{sl} uses two constants $S1$ and $S2$ that can be adjusted to control the duration. $S1$ controls the influence of the ratio between time stealing and working r_s, whereas $S2$ controls the influence of the number of consecutive inefficient quanta n_{iq}:

$$d_{sl} = d_{ql} * r_s * S1 + d_{ql} * n_{iq} * S2 \tag{5.2}$$

When the worker wakes up again, the next quantum is started. By design the worker continues its execution trying to steal a task from another thread.

However, without any further addition there is a flaw in the design of the algorithm similar to the desire instability described in Section 21: Since a worker wakes up just because a certain period of time has passed, there is no guarantee that there are tasks available to be processed by the worker. So in a worst case scenario the worker continuously switches between sleeping and trying to steal tasks without success for a full quantum. In such cases it would be better to reduce the length of the quantum after a period of sleep.

Based on this reasoning, the quantum end time after a period of sleep is adjusted according to the number of inefficient quanta n_{iq}. The more inefficient quanta occurred before, the shorter the next quantum gets:

$$t_{qe} = n + \frac{d_{ql}}{n_{iq} * C + 1} \tag{5.3}$$

C is another constant to control the behavior of the algorithm.

When a quantum is classified as efficient, the next quantum length equals the default quantum length d_{ql}:

$$t_{qe} = n + d_{ql} \tag{5.4}$$

It can still happen that a worker continuously switches between periods of sleep and completely inefficient quanta. However, the more inefficient quanta in succession occur, the longer the sleep periods get and the shorter the quanta between. In Figure 5.2 these adjustments are

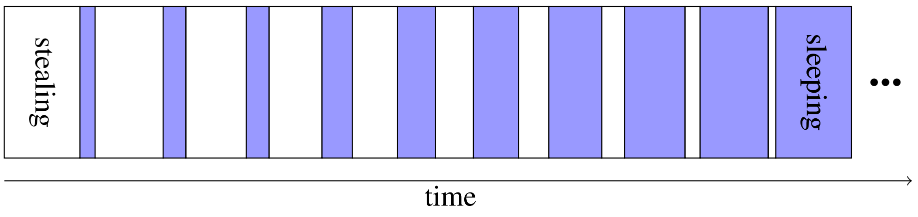


Figure 5.2: Adjustment of quantum length and sleep periods in dynamic quanta algorithm

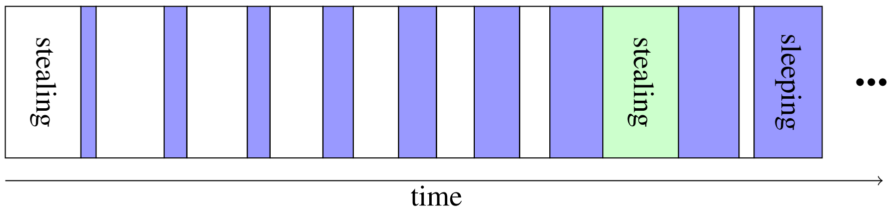


Figure 5.3: Insertion of quantum with normal length in between

shown. In this figure it is assumed that the worker cannot find any tasks to steal, all steal attempts are unsuccessful. The white boxes are periods where the worker tries to steal tasks, the blue boxes symbolize periods where the worker is sleeping. After each inefficient quantum the quantum length is reduced and therefore the time spans where the worker tries to steal tasks get shorter. Contrary the periods where the thread is sleeping are getting longer.

Nevertheless this tweak can lead to a new problem: When there are many inefficient quanta, the quanta between the sleep periods can get very short. This could mean that in this small period of time no task is found, even when there are enough tasks available. So when there are many inefficient quanta, periodically one quantum with normal length is inserted in between. In these longer quanta, the worker has enough time to find a task if there is one available. In Figure 5.3 such inserted quantum with longer length is shown in green.

End-of-quantum after first unsuccessful steal attempt

While performing early test runs of the scheduler, the question how to handle the end of a quantum if it occurs at a first unsuccessful steal attempt, came up. Tests showed that changing the behavior of the algorithm in such cases influences the performance in some test cases. Therefore it was decided to analyze such situations in greater detail.

It might not be a good idea to block a worker after only one failed steal attempt. The basic idea of work-stealing is trying to steal tasks from other workers. Therefore a single failed steal attempt probably is not a good indicator that there are not enough tasks for all workers. So the basic question is: Should it be avoided to put a worker to sleep right after a first unsuccessful steal attempt? For our algorithm this also means: Should the normal `on-quantum-end` procedure be invoked after a first unsuccessful steal?

The time a single failed steal attempt takes is short compared to the time the execution of an average tasks requires. Since a worker is not interrupted when it executes a task, one can assume that a quantum that ends after a first unsuccessful steal attempt, is classified as efficient. This can be seen in Figure 5.4, where quantum Q_1 ends after a first unsuccessful steal attempt. If the normal `on-quantum-end` procedure is invoked between Q_1 and Q_2, it is very likely that Q_1 is an efficient quantum and the worker is not put to sleep.

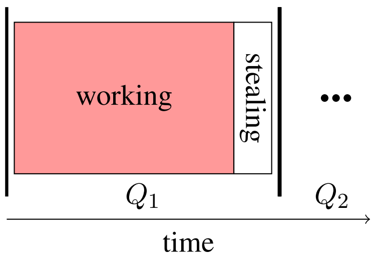


Figure 5.4: First unsuccessful steal attempt after working on tasks

Based on this perception it is questionable, if it is reasonable to handle such situations with the normal `on-quantum-end` function. We therefore came up with different ideas how to handle such situations:

- It is reasonable to argue that the first unsuccessful steal attempt can nevertheless be handled like any other unsuccessful one: The procedure for the end of the quantum is not very expensive, and since the worker has run out of work it may does not make a big difference. In almost all cases the quantum is classified as efficient and the worker continues stealing. Additionally it can also be assumed that the number of inefficient quanta n_{iq} is reset to 0. However, in rare cases it might also be possible that the worker is put to sleep, if the period of work before the first unsuccessful steal attempt was very short.

- Another possibility is to invoke the normal procedure for the end of a quantum, but preventing the worker to be put to sleep. So all counters are adjusted accordingly to the classification of the quantum and the length of the next quantum is adapted. This also means that the number of inefficient quanta n_{iq} is increased if the quantum was inefficient or reset to zero if it was efficient. If it is reset to zero, the information about inefficient quanta is somehow lost without using it.

- Another solution is to simply continue with the current quantum, and defer the checking of the condition for classification of the quantum to the next failed steal attempt. Because a failed steal attempt does not take a lot of time, it is still likely that the quantum later is classified as efficient. Due to the likely classification of the quantum as efficient, again the fact that the recent steal attempts failed, does not change the classification of the quantum.

- Like described the classification of the quantum that ends at a first unsuccessful steal attempt is not very meaningful. Based on this perception it might be reasonable not to take such quanta into consideration at all. The algorithm behaves like such a quantum never happened. In detail this means that it not checked whether the quantum was efficient or inefficient. Additionally the number of inefficient quanta is neither reset nor increased.

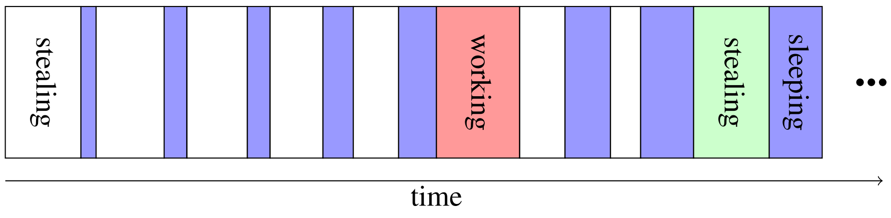


Figure 5.5: Sleep, work and steal periods with scheduler where quantum after first unsuccessful steal attempt is not taken into consideration

> This is important, since this number influences the time to sleep if the next quantum is inefficient. It also influences the length of a quantum after an inefficient quantum.

All the described implementation variations for the first unsuccessful steal attempt after a series of successful steals, were tested. In most cases all of them have a similar performance. However, some of them have bad worst-case scenarios. Overall the last variant where the quantum is not taken into consideration at all, performed the best. Nevertheless it has to be said that there are situations where another of the described alternatives performs better.

Figure 5.5 visualizes the behavior of the chosen variant for handling a first unsuccessful steal attempt. It is assumed that in the red working period exactly one task is executed. After the working period the quantum ends at a first unsuccessful steal attempt. Like described the classification of this quantum is omitted and the scheduler behaves like this quantum never happened. The minimum length of the following quantum (that is the white quantum after the red one) is the same as the length of the preceding quantum. In the figure the green part is a quantum with normal minimum quantum length that is inserted after a series of shorter quantum, like described in the section about the `end-of-quantum` function.

The modified procedure that is invoked after each failed steal attempt can be seen in Algorithm 5.4.

Hierarchical victim selection

The original Pheet work-stealing scheduler that uses the backoff mechanism does not try to back off the thread after each failed steal attempt. It only backs off if no task was found after traversing the whole hierarchy. In Section 4.2 this is described in more detail. In contrast, a quantum in the dynamic quanta algorithm can end after an arbitrary failed attempt to steal a task.

The original principle can also be used in the dynamic quanta algorithm. This would mean that it is only checked if the end of the quantum is reached, after it was tried to steal a task on all levels. If this principle is used, it would not be necessary to insert longer quanta after some inefficient quanta. It would be tried to find a task on all levels, no matter how long the quantum might get.

It was tested if this approach is beneficial for the dynamic quanta algorithm: For this the condition if the end of a quantum is reached is only checked after no task to steal was found on the whole hierarchy. In this case the worker can only go to sleep at this point. However, no evident advantages where found using this approach.

Algorithm 5.4: Modified on-failed-steal function

```
1  if $n_{ua} = 0$ then
       first unsuccessful steal attempt
2      if $t_{qe}$ < Now then
           start new quantum after first unsuccessful steal
           attempt
3          $d_s \leftarrow 0$;
4          $t_{qe} \leftarrow$ Now $+ t_{qe} - t_{qs}$;  minimum length of next quantum equals
           minimum length of previous quantum
5          $t_{qs} \leftarrow$ Now;
6      end
7      $t_{us} \leftarrow$ Now;
8  else
9      if $t_{qe}$ < Now then
10         $d_s \leftarrow d_s +$ Now $- t_{us}$;             update time spent stealing
11         on-quantum-end();
12     end
13 end
14 $n_{ua} \leftarrow n_{ua} + 1$;     update number of unsuccessful steal attempts
```

Randomization

Like in the case of the backoffs, we initially included some randomness into the calculation of the sleep time and quantum length. However, after testing including randomization in the backoff implementations, we also tested our dynamic quanta algorithm if there is really an advantage when some randomization is used. Like in the case of the backoffs, random values do not improve the performance at all. Therefore we removed the randomization for the calculation of the sleep times and quanta lengths. Since randomization is used when selecting victims to steal tasks from, the probability that two workers wake up at the same time and try to steal a task from the same victim is very small. Therefore including additional randomization does not bring an advantage.

5.4 Parameter tuning

An important part of the development of the algorithm is choosing good default values for the parameters of the algorithm:

- Default quantum length d_{ql}
- The constants for the calculation of sleep times $S1$ and $S2$
- The constant for the calculation for the quantum length after a period of sleep C

Variant	$S1$	$S2$
dynquanta_a	0.03	0.06
dynquanta_b	0.3	0.6
dynquanta_c	0.003	0.006
dynquanta_d	0.1	0.2

Table 5.1: Tested constants $S1$ and $S2$ for dynamic quanta algorithm

- The threshold T that determines when workers are suspended.
- The maximum sleep time, which prevents workers to sleep for a too long time.

Constants $S1$ and $S2$ to calculate sleep time

The longer the workers are suspended (sleep times), the fewer cycles are wasted when there are not enough tasks for all workers. However, it may also be that during the period of sleep many new tasks are spawned, which could be executed earlier if the workers were not suspended. Therefore it is a trade-off between trying to waste as few cycles as possible and not missing phases of high parallelism in the application.

After some initial experimental small test runs, we decided to test four different scheduler variants that are shown in Table 5.1. We tested $S1$ values between 0.003 and 0.3 and $S2$ values between 0.006 and 0.6. For these tests we set the quantum length d_{ql} to 0.3ms, the maximum sleep time to 1.5ms, the threshold T to 0.5 and C to 2.0.

The test results in Figure 5.6 show that values of 0.003 and 0.006 for $S1$ and $S2$ are clearly too small in most benchmarks. The test results for the other variants are quite similar, we therefore chose 0.1 for $S1$ and 0.2 for $S2$ (dynquanta_d) as constants which are used for further tests.

Maximum duration of sleep phases

Since changing the maximum duration of sleep does not influence the performance of the backoff a lot, we assumed that this is also the case for the dynamic quanta algorithm. The tests showed very similar results like the tests for the backoff versions: It is harmful to set a very small maximum sleep time and therefore a large enough value has to be chosen. The performance does not change significantly, as long as the value is reasonable large.

Figure 5.7 shows the results for different maximum sleep times for the quicksort benchmark on Mars. One can see that in this benchmark all values that use a maximum sleep time of 1.0ms and larger perform very similar. Maximum sleep times that are smaller than 0.5ms worsen the performance significantly when all 80 cores are used. According to the results of this benchmark it would even be reasonable not to use a maximum sleep time at all. However, in some of the benchmarks the scheduler without a maximum sleep time performs slightly worse than the ones with a capped sleep time. After analyzing all benchmark results, we decided to use a value of 1.5ms as default value for the maximum sleep time.

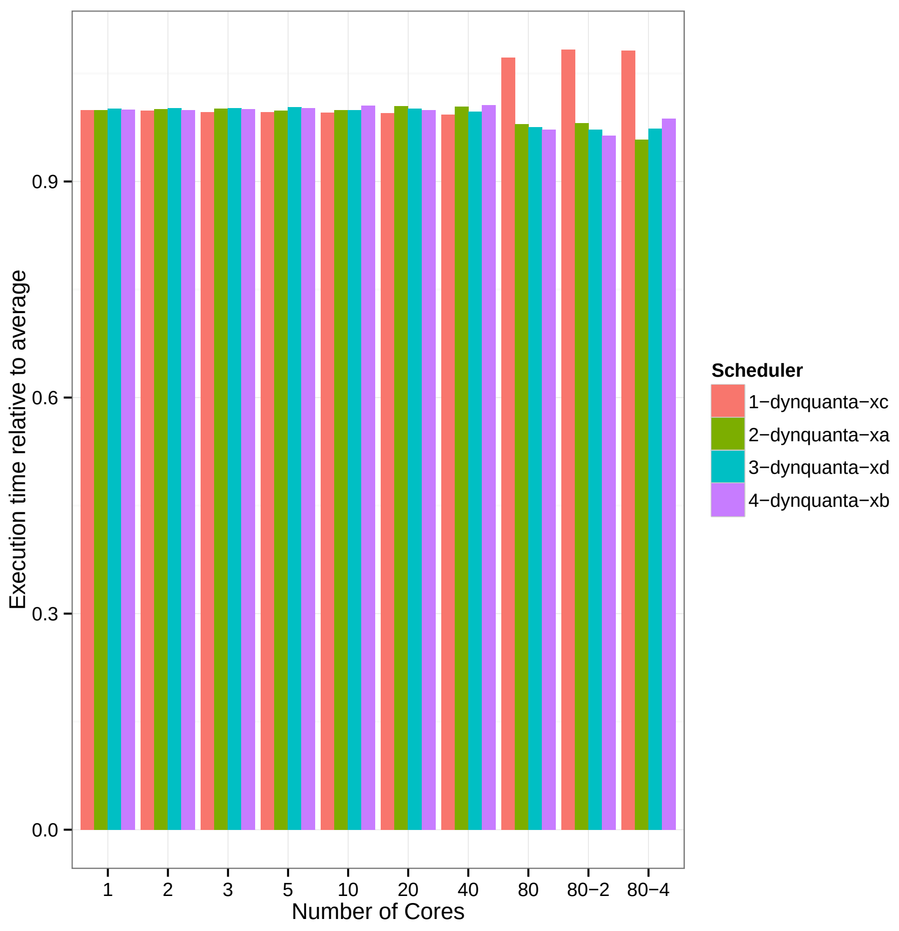


Figure 5.6: Aggregated results showing impact of constants $S1$ and $S2$ on Mars

Quantum Length d_{ql}

Short quanta mean higher overhead for checking the adaptive conditions more often and may lead to situations where threads constantly switch between sleeping and working. Longer quanta lead to lower overhead and may smooth out highly variable degrees of parallelism. However, they could also miss important changes in the degree of parallelism that would need quick switching between sleeping and working states.

Cao et al. [15] use a quantum length of 10 milliseconds in their evaluation of the Stable Adaptive Work-Stealing algorithm presented in Section 2.4. Evaluations of our dynamic quanta algorithm showed that much shorter quanta are beneficial in our case. The reason for this is that the algorithm is much more lightweight and does not induce a lot of overhead. Additionally since only local information of each worker is used, the predictions for the resource usage are probably not as good. Therefore shorter periods that allow faster adaption to varying degrees of parallelism are needed.

We tested quantum lengths between 16us and 600us. Since adjusting the quantum length

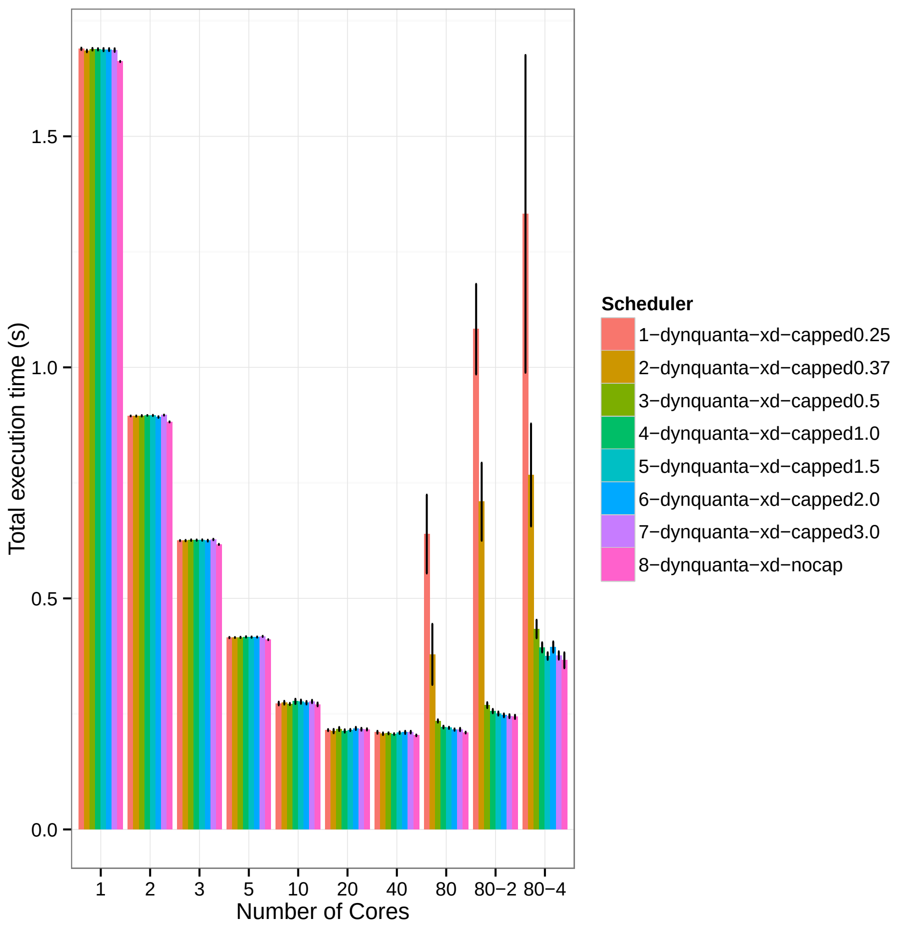


Figure 5.7: Results showing impact of maximum sleep time in quicksort benchmark on Mars

directly influences the duration of the sleep phases, we also adjusted the sleep constants $S1$ and $S2$ in these tests: The base for the tests is a quantum length of 300us with $S1$ and $S2$ values of 0.1 and 0.2 respectively. When the quantum length is increased, the S-constants are decreased by the same factor. For example when the quantum length is doubled, both S-constants are halved. This lead to a set of parameter values that we tested and which are shown in Table 5.2. For the tests the maximum sleep time is set to 1.5ms, the threshold T to 0.5 and C to 2.0.

The test results on Mars are shown in Figure 5.8. One can see that the versions with a short quantum length perform slightly better than the versions that use a longer quantum length. However, there are some benchmarks where the very short quanta of length 16us and 33us are worse than the slightly larger ones. Therefore we chose a default minimum quantum length d_{ql} of 75us. The default values for $S1$ and $S2$ are set to 0.4 and 0.8 respectively.

Variant	quantum length d_{ql} (us)	$S1$	$S2$
dynquanta_d	300	0.1	0.2
dynquanta_e	600	0.05	0.1
dynquanta_f	150	0.2	0.4
dynquanta_g	75	0.4	0.8
dynquanta_h	33	0.8	1.6
dynquanta_i	16	1.6	3.2

Table 5.2: Variants to determine best default quantum length d_{ql} for dynamic quanta algorithm

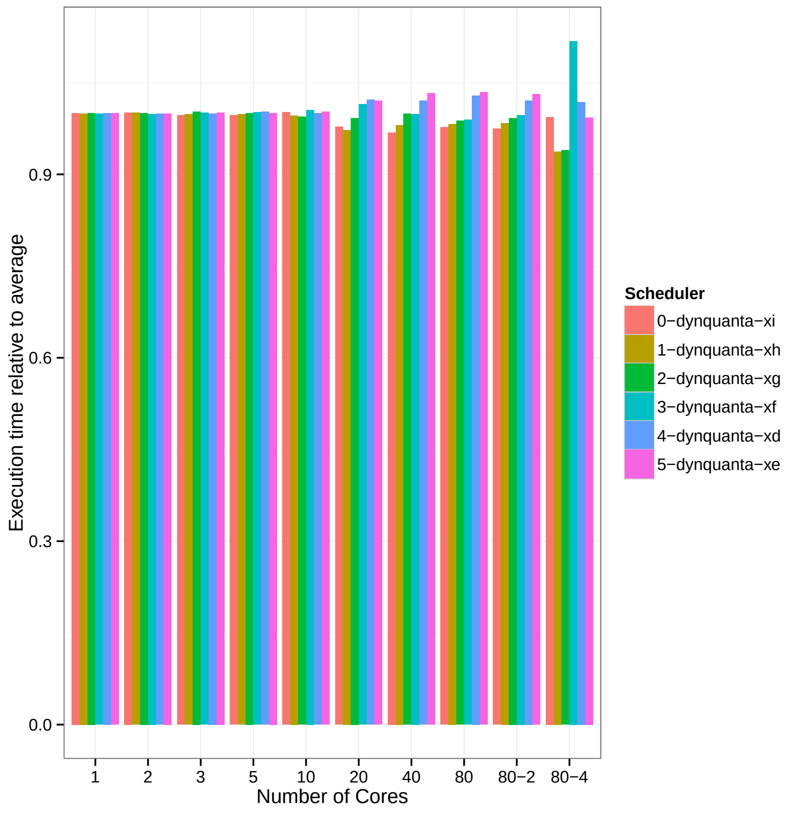


Figure 5.8: Aggregated results that show impact of default quantum length d_{ql} parameter on Mars

Parameter	**Value**	**Influence on performance**
$S1$	0.4	medium
$S2$	0.8	medium
Maximum sleep duration	1.5ms	great
Quantum length d_{ql}	75us	medium
Threshold T	0.5	small
Constant C	2	small

Table 5.3: Chosen parameter values for dynamic quanta algorithm

Threshold T for classification of quanta

For the threshold T, which categorizes a quantum as efficient or inefficient, Agrawal et al. state that typical values might be in the range of 80% to 95% [4]. Since the dynamic quanta algorithm is more fine-grained and putting a worker to sleep does not have that much overhead, it is expected that a smaller threshold is a better choice for the algorithm. In fact the value of the threshold is not that important in our algorithm: Since only local information is used, most of the time r_s is either very small when the worker has tasks to process or very high when it cannot find a task to steal. So almost always the distinction between inefficient and efficient quanta is pretty obvious and changing the threshold T does not influence the behavior or performance a lot.

We performed tests with a threshold of 0.3, 0.5 and 0.7. As expected no significant performance differences occurred using these three different values for the threshold. We therefore chose 0.5 as default value for the threshold.

Constant C to adjust quantum length

The constant C influences the length of a quantum after a worker was blocked by putting it to sleep. The greater C the smaller the duration of such quanta get. We tested values between 0.2 and 20 to find out how big the influence this adjustment of the quantum length is on the performance of the scheduler. In our benchmarks we did not see any significant differences between these tested C values. It seems like that the adjustment of the quantum length does not influence the performance at all or just to a small degree. We decided to use 2 as default value for C.

Conclusion parameter tuning

The final parameter values for the dynamic quanta algorithm can be seen in Table 5.3. The table also contains a column that states how big the influence of the parameter is on the performance of the scheduler.

The tests showed that the most important parameter value is the maximum sleep duration. It must be ensured that this duration is large enough, otherwise the performance suffers significantly when all cores are used, especially in multiprogrammed environments. The other parameters of the algorithm do not change the performance of the scheduler that much.

One interesting fact is that the best minimum quantum length d_{ql} is set to 75us which is rather short. This means that after processing only one task in a quantum, the quantum will probably be classified as efficient.

5.5 Long and short quanta algorithm

Since the dynamic quanta algorithm uses a quite small quantum length, brief events like processing a single task have a great influence on the quantification of the quantum as efficient or inefficient. Therefore the decision whether a worker is put to sleep or not is also quickly affected by such events. According to the benchmarks these short quanta are useful, however it is interesting what happens if additionally measurements over a longer timespan are taken into consideration.

Introduction long and short quanta algorithm

In our *long and short quanta* algorithm the basic ideas of the dynamic quanta algorithm stay the same. There still exists the idea of quanta, which do not have a fixed length but only a minimum length. Additionally to the relatively short dynamic quantum a second type of quantum is used. This other type of quantum is longer and also records the times that are spent stealing and working. We call this new type of quantum *long quantum*.

When a worker is put to sleep, due to the data measured in the short quantum, the ratio between work and steal time in the previous long quantum influences the duration how long the thread is put to sleep. Workers can only be put to sleep between short quanta and not between long quanta. The sole purpose of long quanta is to provide more information how long a worker should be blocked when it is decided that there are not enough tasks available. Long quanta provide a more long-term view of the utilization of the worker.

To also take such long quanta into consideration, the procedures of the dynamic quanta algorithm were modified. The modified procedures for the long quanta algorithm are presented on the next few pages.

Detailed algorithm

The long and short quanta algorithm needs an additional set of variables that store basically the same information like the ones in the basic dynamic quanta algorithm. All variables that store information about the long quantum have an l in their name.

Algorithm 5.5 shows the modified function that is called after a failed steal attempt. It uses the following additional set of variables:

Long quantum start time t_{lqs} This time point contains the start time of the current long quantum.

Long quantum end time t_{lqe} A time point that defines until when the current long quantum at minimum lasts.

Long quantum steal time d_{ls} Duration how long so far all steal attempts in the current long quantum lasted.

Long quantum unsuccessful steals start time t_{lus} Time point at which the last unsuccessful steal attempt occurred that until now was not followed by a successful one. It is needed to calculate the long quantum steal time. So it is the time of the first unsuccessful steal attempt in the long quantum.

Like for the short quantum, the quantum end time of the long quantum has to be checked and the time spent stealing in the long quantum has to be adjusted. If a long quantum has ended, a separate function `on-long-quantum-end` is invoked. This procedure is shown further below.

After a successful steal the time spent stealing in the long quantum has to be increased. This is shown in Algorithm 5.6.

The gathered data for the long quantum is used in the procedure for the end of the short quantum, which is presented in Algorithm 5.7. It is very similar to the basic dynamic quanta algorithm as it was shown in Algorithm 5.3. The differences are mainly the extensions for the calculation of the data for the long quanta.

In the procedure the ratio between the times spent stealing and working in the long quantum influences the timespan a worker is put to sleep after an inefficient short quantum. However, not only the ratio in the current long quantum is used but also the one from the previous long quantum. For this purpose each worker stores an additional variable:

Ratio steal/work time of previous long quantum r_{pls} Before a new long quantum is started, the ratio between the time spent stealing and working in the current long quantum is stored in this variable. Therefore this variable contains the ratio between time spent stealing and working in the previous long quantum.

This is necessary since the current long quantum may be running for a very short time when the short quantum ends. In other words, the end of a short quantum can potentially be very shortly after the beginning of a long quantum. So the work and steal times in the current long quantum may not be very helpful. Therefore the arithmetic mean between the ratio in the current and in the previous long quantum is used:

$$r_{ls} = \frac{\frac{d_{ls}}{n - t_{lqs}} + r_{pls}}{2} \tag{5.5}$$

In Procedure 5.7 this calculated ratio is stored in the temporary variable r_{ls}. This ratio is then used for the calculation of the sleep duration of the worker:

$$d_{sl} = d_{ql} * r_s * r_{ls} * S1 + d_{ql} * n_{iq} * S2 \tag{5.6}$$

After a period of sleep the sleep duration is added to the long quantum start time t_{lqs}. This is done to exclude the sleep periods from the calculation of the steal/work ratio of the long quantum. These periods count neither as stealing nor working.

Algorithm 5.5: Long quanta algorithm on-failed-steal function

```
if $n_{ua} = 0$ then
    first unsuccessful steal attempt
    if $t_{qe}$ < Now then
        start new quantum after first unsuccessful steal attempt
        $d_s \leftarrow 0$;
        $t_{qe} \leftarrow \text{Now} + t_{qe} - t_{qs}$;  minimum length of next quantum equals minimum length of previous quantum
        $t_{qs} \leftarrow \text{Now}$;
    end
    if $t_{lqe}$ < Now then
        on-long-quantum-end();
    end
    $t_{us} \leftarrow \text{Now}$;
    $t_{lus} \leftarrow \text{Now}$;
else
    $shortEnd \leftarrow \perp$;
    if $t_{qe}$ < Now then
        $d_s \leftarrow d_s + \text{Now} - t_{us}$;      update time stealing short quantum
        $d_{ls} \leftarrow d_{ls} + \text{Now} - t_{lus}$;      update time stealing long quantum
        on-quantum-end();
        $t_{lus} \leftarrow \text{Now}$;
        $shortEnd \leftarrow \top$;
    end
    if $t_{lqe}$ < Now then
        if $\neg shortEnd$ then
            $d_{ls} \leftarrow d_{ls} + \text{Now} - t_{lus}$;   update time stealing long quantum
        end
        on-long-quantum-end();
    end
end
$n_{ua} \leftarrow n_{ua} + 1$;     update number of unsuccessful steal attempts
```

Algorithm 5.6: Long quanta algorithm on-successful-steal function

```
if $n_{ua}$ > 0 then
    $d_s \leftarrow d_s + \text{Now} - t_{us}$;        update time stealing short quantum
    $d_{ls} \leftarrow d_{ls} + \text{Now} - t_{lus}$;        update time stealing long quantum
    $n_{ua} \leftarrow 0$;
end
```

Algorithm 5.7: Long quanta algorithm on-quantum-end function

```
1  $d_q \leftarrow \texttt{Now} - t_{qs}$;          calculates actual length of short quantum
2  $r_s \leftarrow d_s/d_q$;                   calculates ratio of time spent stealing
3  $d_lq \leftarrow \texttt{Now} - t_{lqs}$;         calculates actual length of long quantum
4  $r_{ls} \leftarrow \frac{d_ls/d_lq+r_{pls}}{2}$;             calculates ratio of time spent stealing
5  if ConditionInefficientQuantum then
       quantum is classified as inefficient
6      $st \leftarrow ...$;                               calculate time to sleep
7      Sleep($st$);                              block execution of thread
8      $t_{lqs} \leftarrow t_{lqs} + st$;          exclude sleep time from long quantum
9      $n_{iq} \leftarrow n_{iq} + 1$;
10     $t_{qe} \leftarrow \texttt{Now} + \frac{d_{ql}}{n_{iq}*C+1}$;         calculate end time of next quantum
11 else
       quantum is classified as efficient
12     $n_{iq} \leftarrow 0$;
13     $t_{qe} \leftarrow \texttt{Now} + d_{ql}$;           calculate end time of next quantum
14 end
15 $d_s \leftarrow 0$;
16 $t_{qs} \leftarrow \texttt{Now}$;
17 $t_{us} \leftarrow \texttt{Now}$;
```

Algorithm 5.8: Long quanta algorithm on-long-quantum-end function

```
1 $r_{pls} \leftarrow d_{ls}/(\texttt{Now} - t_{lqs})$;
2 $t_{lqs} \leftarrow \texttt{Now}$;
3 $t_{lus} \leftarrow \texttt{Now}$;
4 $t_{lqe} \leftarrow \texttt{Now} + d_{lql}$;
5 $d_{ls} \leftarrow 0$;
```

The procedure for the end of a long quantum shown in Algorithm 5.8 does not do a lot of work. It calculates the ratio between the time stealing and working and stores it in the variable r_{pls}. It then resets all variables that are used for the long quantum. The minimum length of the next long quantum is always set to the default minimum length of the long quanta d_{lql}. So the minimum length of a long quantum is constant and not dynamically adjusted like the one of a short quantum.

Condition for classification of quantum

There are a few possibilities for the classification of a quantum as efficient or inefficient, the condition is denoted as *ConditionInefficientQuantum* in Algorithm 5.7:

- Only the ratio in the current short quantum can be used. This is the same like in the basic

Parameter	**Value**
condition inefficient quanta	$r_s > T \lor r_{ls} > T$
short quantum length d_{ql}	75us
long quantum length d_{lql}	3ms
Threshold efficient/inefficient quantum T	0.6
Sleep constant 1 S_1	0.8
Sleep constant 2 S_2	0.8
Quantum length constant C	2.0
Maximum sleep time	1.5ms

Table 5.4: Chosen parameter values for dynamic long quanta algorithm

dynamic quanta algorithm: $r_s > T$. The long quanta are therefore only used to calculate the sleep duration after an inefficient quantum.

- The quantum is classified as inefficient if both the ratio in the short *and* in the long quantum are above the threshold T: $r_s > T \land r_{ls} > T$. This implies that fewer quanta are classified as inefficient and therefore workers are less often put to sleep.

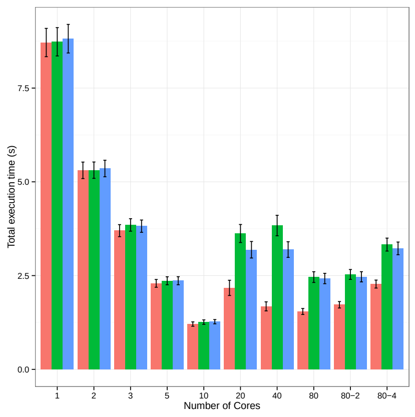
- Contrary to the previous option, a quantum can be classified as inefficient if the ratio in the short *or* in the long quantum is above the threshold: $r_s > T \lor r_{ls} > T$. This has the converse effect that more quanta are categorized as inefficient.

- In sort of in-between the last two solutions is to use the arithmetic mean of both ratios: $(r_s + r_{ls})/2 > T$. Using this condition the threshold T becomes more important. In phases where the state of the worker is switching from working (enough tasks) to stealing (no tasks) and the other way round, it can happen that one ratio is nearly 1 and the other ratio almost 0. Therefore the mean of these two ratios is around 0.5. Adjustments of the threshold T may influence the behavior of the scheduler more than when using one of the other conditions or the basic dynamic quanta algorithm.

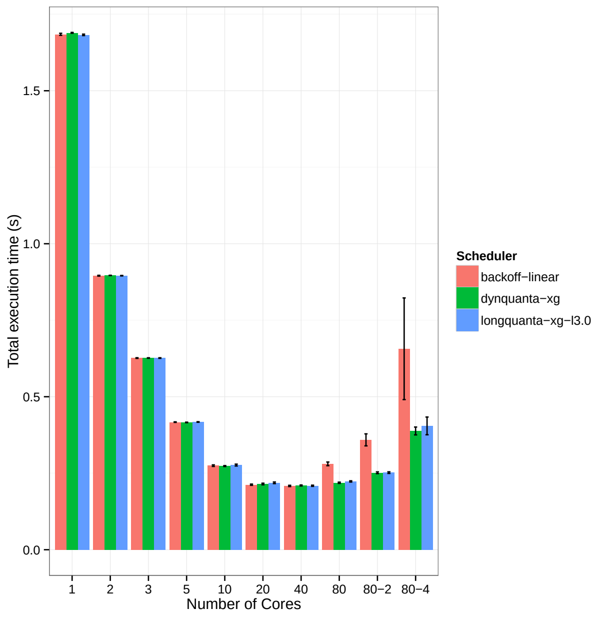
Parameter selection

Since there are a lot of parameters that can be changed and tested, we decided to use the parameters selected for the basic dynamic quanta algorithm also for the long and short quanta algorithm. We then tested which variant for the condition whether a quantum is efficient or inefficient is the best. The differences between this variants were not very large, we decided to use the condition $r_s > T \lor r_{ls} > T$ for further tests.

Since both steal ratios of the short and the long quantum influence the sleep duration, we set $S1$ to 0.8 instead of 0.4 for the following benchmarks. We tested durations for the long quanta between 0.3ms and 5.0ms and values for the threshold T between 0.4 and 0.8. Table 5.4 shows the best parameter values we found for the long quanta algorithm.



(a) Graph benchmark on Mars



(b) Quicksort benchmark on Mars

Figure 5.9: Results for quicksort and graph benchmarks on Mars

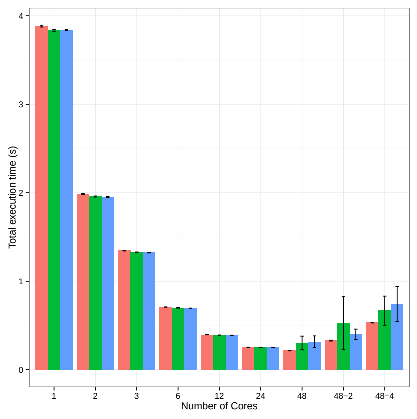
Testing the different parameter values for the long dynamic quanta algorithm revealed that the performance of this algorithm is not significantly better than the basic dynamic quanta algorithm. In some tests the extended algorithm is better, in some worse than the basic one. Overall there are no evident benefits from the introduction of long quanta. Since this version needs a little bit more memory for storing the status information of the long quanta, we choose to spend not that much time optimizing the parameters like for the basic dynamic quanta algorithm. An overview of the performance of both adaptive algorithms is given in the next section.

5.6 Performance Tests

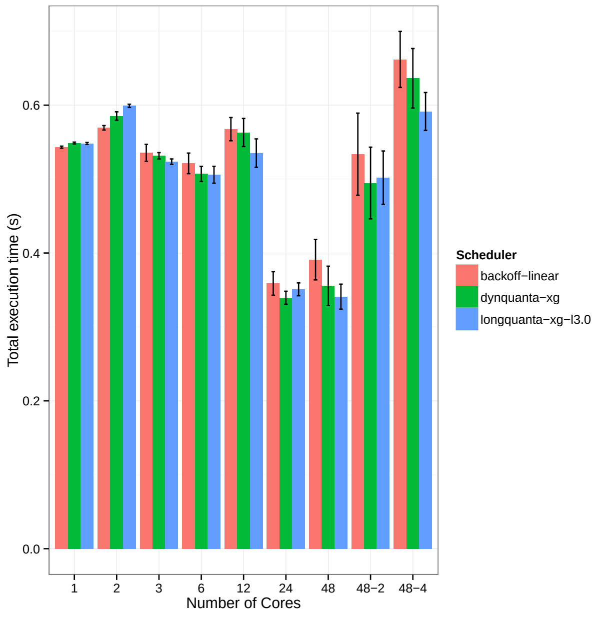
In this Section a performance overview of the dynamic quanta algorithm and the long quanta algorithm is given. The parameters used for the two algorithms are the already presented ones in Table 5.3 and Table 5.4. For comparison the figures also show the results of the scheduler using the linear backoff that was presented in the Section about backoff techniques.

Figure 5.9 shows the results of two benchmarks on Mars: Figure 5.9a shows the results for the graph bipartitioning benchmark, Figure 5.9b for the quicksort benchmark.

In the graph bipartitioning benchmark the scheduler using the linear backoff performs significantly better than the adaptive schedulers, when more than 10 cores are used. On the other hand, in the Quicksort benchmark the adaptive algorithms outperform the linear backoff, when all 80 cores are used and in the multiprogrammed environment tests. Between the two adaptive algorithms, sometimes the basic algorithm is faster and sometimes the long and short quanta al-



(a) UTS on a geometric tree benchmark on Saturn



(b) Single-source Shortest Path benchmark on Saturn

Figure 5.10: Results for UTS and SSSP benchmarks on Saturn

gorithm. The other benchmark results on Mars reveal that sometimes the linear backoff is faster and sometimes the adaptive algorithms. However, there are more cases where the simple linear backoff algorithm outperforms the adaptive ones. In total no significant differences in terms of performance exist between the two adaptive algorithms.

Figure 5.10 shows the results of two benchmarks on Saturn: Figure 5.10a shows the results for the Unbalanced Tree Search on the geometric tree, Figure 5.10b for the Single-source Shortest Path benchmark.

Like on Mars, also on Saturn benchmarks where the adaptive algorithms improve performance and benchmarks where the performance is worse, exist. For example in the UTS benchmark the linear backoff performs better, whereas in the Single-source Shortest Path benchmark the adaptive algorithms perform slightly better. In general, the performance differences on Saturn are not so big like on Mars. Overall the same conclusion like for the results on Mars can be drawn, in more benchmarks the linear backoff outperforms the adaptive algorithms than vice versa. The two adaptive algorithms roughly perform the same.

5.7 Comparison with backoff techniques

After presenting and testing both the scheduler with an optimized backoff and the two adaptive algorithms, in this Section we try to compare the two approaches. Before we started this work, we hoped to be able to design and implement an adaptive scheduler with the presented design goals that is able to outperform schedulers that use a simple backoff. In a very early stage of this

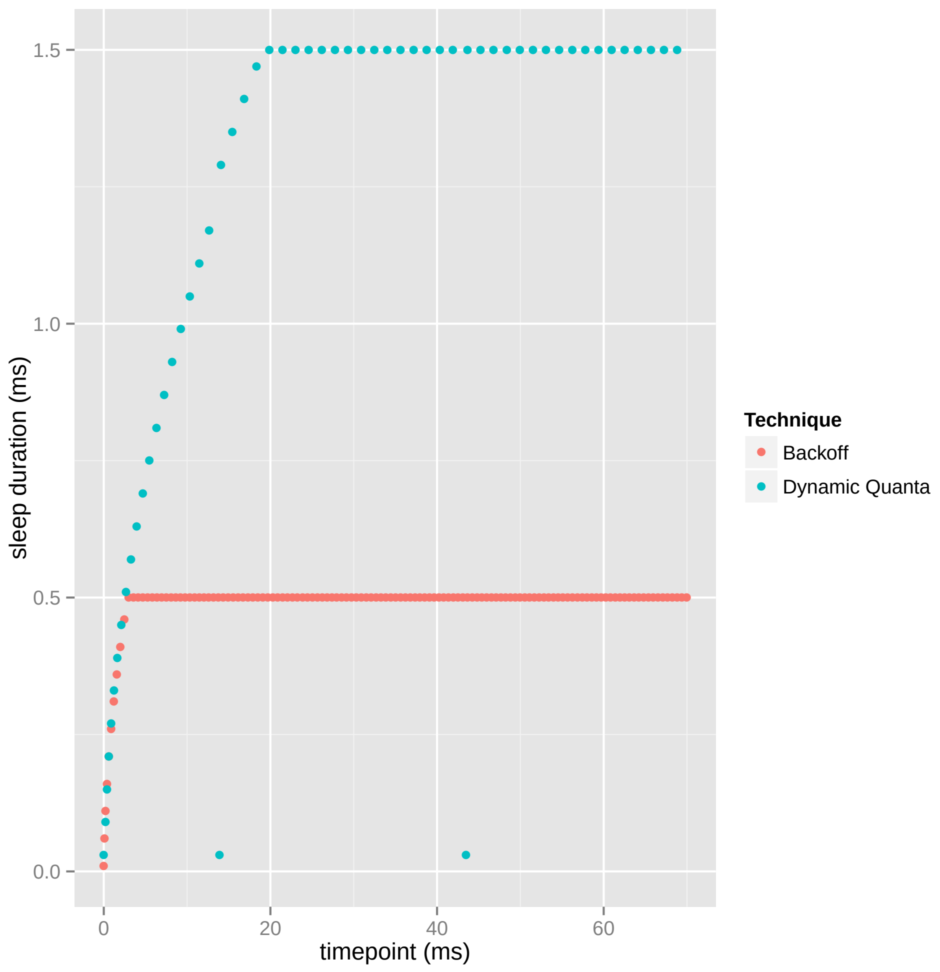


Figure 5.11: Theoretical sleep times of backoff and dynamic quanta algorithm in a synthetic situation where no tasks to steal exist

work, we even reached this goal: The first implementations of the backoffs performed clearly worse than the first implementations of the adaptive algorithms. However, after comprehensive work on optimization of the parameters of the backoffs, the scheduler using the simple linear backoff is not worse, but better than our adaptive algorithms. Tuning the parameters of the adaptive algorithms improved their performance as well, however not in this extent.

Therefore in this Section we try to compare the scheduler using the linear backoff with the scheduler that uses our dynamic quanta algorithm.

First of all we analyze the differences in behavior in a simulated situation. Figure 5.11 shows such a situation, where a worker cannot find a task to steal. One can see that the characteristics of the two methods are very alike, the sleep time is increased in a very similar way. Since we optimized the parameters of both the backoff and the dynamic quanta algorithm, we assume that this is some kind of optimal strategy to increase the sleep duration. In the dynamic quanta algorithm the sleep duration is capped at 1.5ms, whereas in the backoff variant the maximum sleep time is 0.5ms. One can see that the dynamic quanta algorithm inserts short periods of sleep from time to time, which are the blue dots at around 13ms and 43ms.

Additionally in this graph the periods between the points of time where the dynamic quanta algorithm puts the worker to sleep are significantly larger than the periods in the backoff implementation. This does not show a characteristic of the scheduler itself. The reason for this is the way this graph was generated: The backoff implementation does not wait between consecutive backoff calls and always blocks the thread. In a real scenario this would depend on the time it takes trying to steal tasks on all levels, according to the hierarchical victim selection policy. On the contrary the dynamic quanta algorithm uses a timer to determine the point of time when a thread is blocked again.

So the time to sleep is increased in a very similar, almost identical way. The obvious difference is that the maximum sleep duration of the adaptive algorithm is 3x larger. However, since also a maximum sleep duration of 0.5ms for the dynamic quanta algorithm was tested, this cannot be the reason for the performance differences.

We also measured the same times in a real benchmark, the quicksort benchmark with an increased problem size of $50.000.000$ elements to sort, on Mars running on 80 cores. This is a benchmark where the dynamic quanta algorithm actually performs better than the backoff version. Measuring these times probably worsens the performance of the scheduler noticeable, but the times are still useful for a general analysis of the behavior. Figure 5.12 shows the sleep durations and time points when the sleep phases occurred of one worker. The data points of the backoff algorithm are not very surprising: Over the whole time of execution the backoff time iteratively is increased until its maximum, sometime later a task to steal is found and the backoff time is reset. The data of the dynamic quanta algorithm is a bit surprising in the first place. It is a little bit unexpected that the sleep duration in the dynamic quanta algorithm is only reset four times. This means that in this worker only four quanta are classified as efficient, all others are inefficient quanta.

One reason for this is that if a quantum ends with a failed steal that occurs after a successful steal, a new quantum is started but the counter for the number of consecutive inefficient quanta n_{iq} is not reset. So when the next quantum again is an inefficient one, the sleep duration is further increased due to the influence of n_{iq} to the sleep duration. Alternatives to this approach were described in Section 5.3. This does not mean that the worker is not processing any tasks at all or just in these four efficient quanta. Tasks can also be processed in inefficient quanta. However, this behavior is an indication that the worker spends a lot of time trying to steal tasks and that many steal attempts fail. Looking at the performance results in Figure 5.9b one can see that the Quicksort benchmark does not benefit from using more than 20 cores. Therefore actually it is really nice that the dynamic quanta algorithm classifies so many quanta as inefficient. Obviously this benchmark does not benefit from the usage of all 80 cores.

Table 5.5 shows a comparison of the total duration of all sleep phases of all workers and the number how often workers were put to sleep. In the dynamic quanta implementation workers are not put to sleep as often as in the backoff implementation. The sleep duration of the dynamic quanta algorithm is sometimes greater and sometimes lower than in the scheduler using the

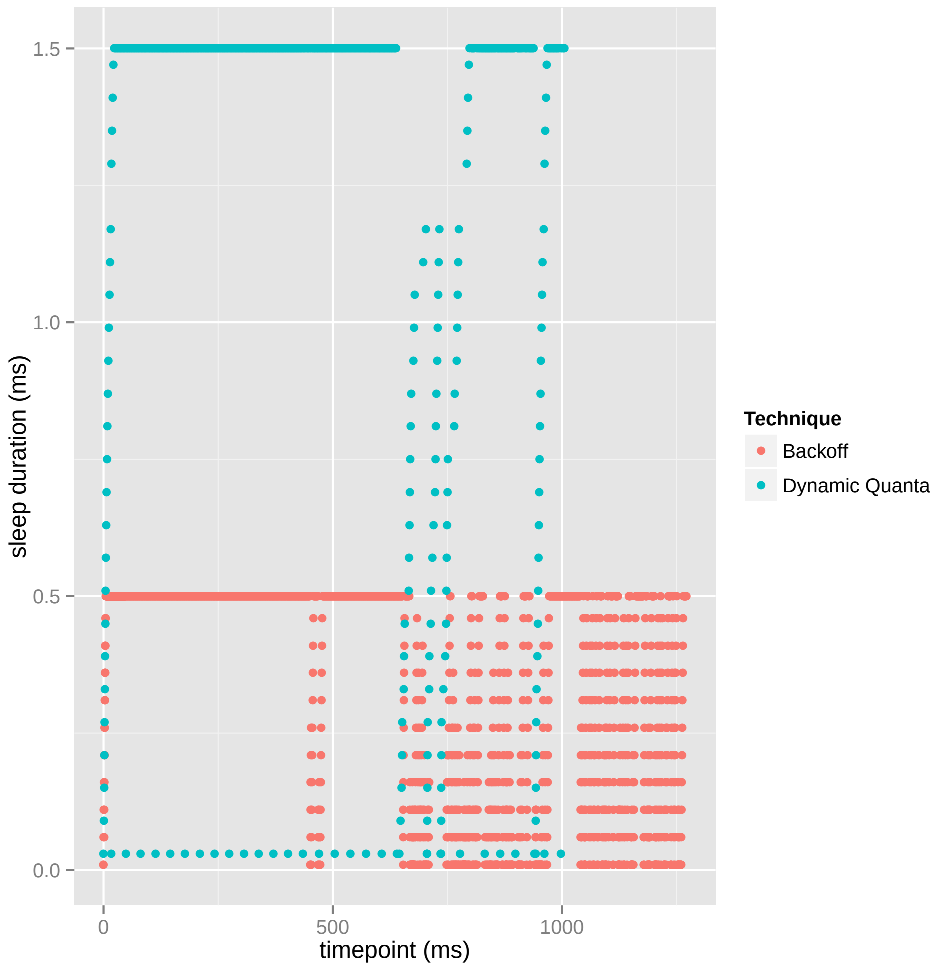


Figure 5.12: Measured duration of sleep phases in dynamic quanta and linear backoff algorithm in sorting benchmark on Mars

Benchmark	Dynamic Quanta	Backoff
LU decomposition	121.26s (81, 258)	112.23s (180, 209)
Quicksort	14.48s (11, 904)	18.35s (36, 147)
Graph Bipartitioning	1.00s (1, 438)	2.09s (3, 789)

Table 5.5: Total sleep duration in seconds and number of times workers were put to sleep for dynamic quanta and backoff implementation

backoff. The probable reason for this is that the adaptive algorithm uses a larger maximum sleep time. However, the absolute differences in the sleep times are not very large.

CHAPTER 6

Benchmark results

In this chapter comparisons of some of schedulers that were developed in the context of this work are given for all executed benchmarks. The chosen schedulers include the ones that overall perform the best and the ones where we think the detailed results are interesting to present. The comparisons additionally contain the test results for the original work-stealing scheduler of Pheet and the modified scheduler that does not use adaptive techniques or a backoff at all.

In this chapter the following schedulers are compared:

linear-backoff-scheduler Scheduler that uses our optimized linear backoff which is described in Chapter 4 in great detail.

dynquanta-scheduler This scheduler uses the basic dynamic quanta algorithm that is described in Chapter 5.

longquanta-scheduler This scheduler denotes the extension of the dynquanta-scheduler, which uses the long and short quanta algorithm that is described in Section 5.5.

noop-scheduler The noop-scheduler does not block its threads at all and always tries to steal tasks until the execution of all tasks is finished.

original-scheduler This scheduler uses the backoff that is currently used by default in Pheet. This backoff is described in Chapter 4.

yield-backoff-scheduler This scheduler uses a backoff that just calls the `yield` system function whenever a thread backs off. It is very similar to the original ABP work-stealing algorithm. However, in the ABP algorithm `yield` is called after each unsuccessful steal attempt, whereas in our yield-backoff-scheduler this depends on the hierarchical victim selection of Pheet. This is described in more detail in Section 4.2

yield-abp-scheduler The yield-abp-scheduler closely resembles the original ABP work-stealing algorithm: Like a scheduler using the ABP algorithm this scheduler calls `yield` after *each* failed steal attempt. However, like all the presented schedulers it still uses the hierarchical victim selection policy. We show tests results of this scheduler, because new work-stealing schedulers are often compared with a classic ABP work-stealing scheduler.

In some of the benchmarks a slowdown occurs when all cores are used. One can ask if this slowdown occurs because *all* or because *so many* cores are used. In our graphs we do not show results for situations where more than half of the cores, but not all cores are used. Therefore the graphs offer no answer to this question. However, we also executed some test runs for these situations and found out that the latter is true: The slowdown does not occur because all cores are used, but also occurs beforehand. For reasons of clearness we did not include these test results in the graphics.

6.1 Graph Bipartitioning

The results of the graph bipartitioning benchmark on Mars can be seen in Figure 6.1. Overall one can say that the linear-backoff-scheduler and the two schedulers that use yield perform the best in the graph bipartitioning benchmark on Mars. The noop-scheduler performs the worst. The dynamic quanta algorithms do not perform very well in this benchmark and are even outperformed by the scheduler using the original backoff. However they are still better than the noop-scheduler.

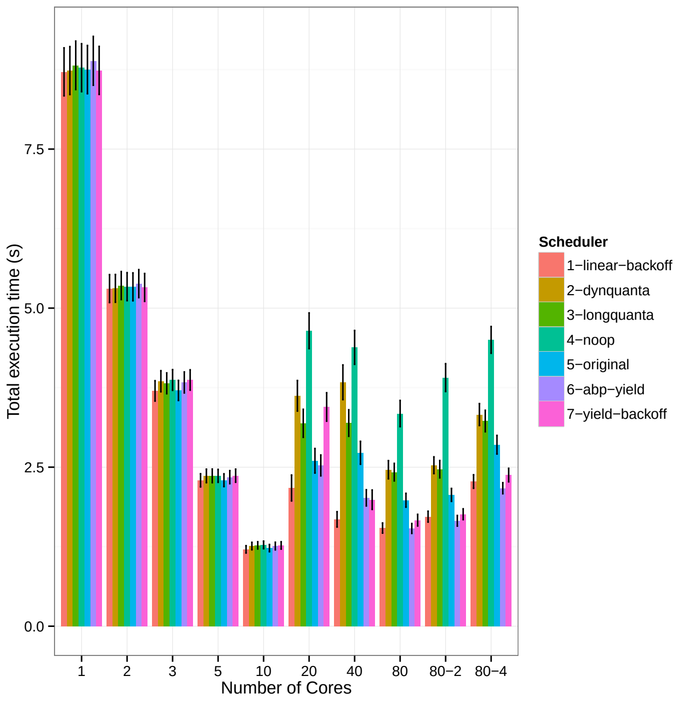


Figure 6.1: Results of Graph Bipartitioning benchmark on Mars

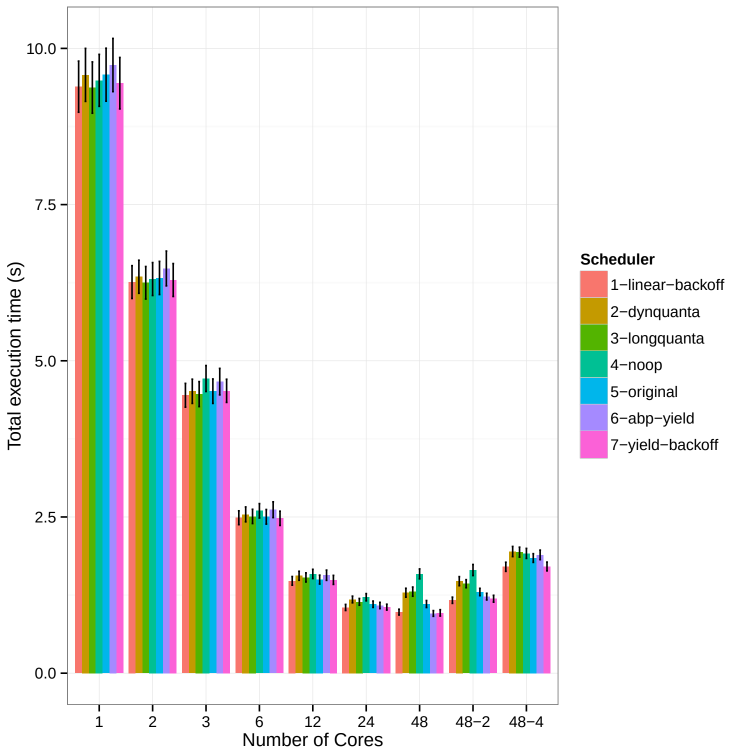


Figure 6.2: Results of Graph Bipartitioning benchmark on Saturn

The results of the benchmark runs on Saturn are shown in Figure 6.2. On Saturn the differences in performance between the schedulers are not very large, whereas they are more significant on Mars. On Mars the liner-backoff-scheduler is almost two times faster than the noop-scheduler, on Saturn the noop-scheduler does not perform that bad.

6.2 LU decomposition

In the LU decomposition benchmark huge differences in performance are visible, which is shown in Figure 6.3: On Mars the original-scheduler, the yield-backoff-scheduler and the abp-yield-scheduler perform really badly when all 80 cores are used. When four instances of the benchmark are running in parallel, the linear-backoff-scheduler finishes the tasks about 18x faster than the original-scheduler. In the test runs for multiprogrammed environments the noop-scheduler is also worse than the other three schedulers. There is no big difference between the linear-backoff-, dynquanta- and longquanta-scheduler on Mars.

On Saturn the results, which are shown by Figure 6.4, are slightly different: Like on Mars in multiprogrammed environments the original-scheduler performs significantly worse than the other schedulers. However, whereas on Mars the yield-backoff-scheduler performs nearly as bad as the original-scheduler, on Saturn the yield-backoff-scheduler is the best scheduler together

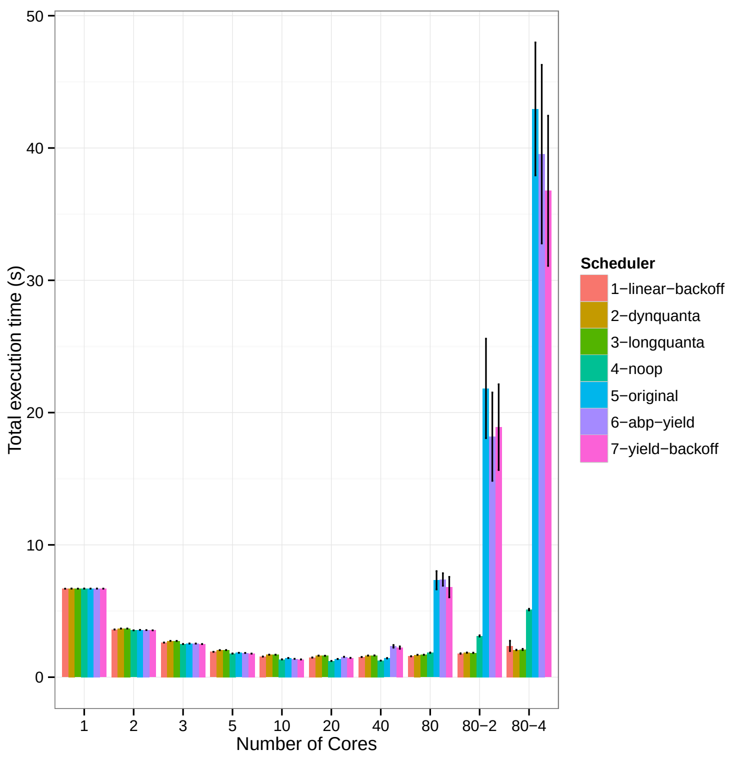


Figure 6.3: Results of LU decomposition benchmark on Mars

with the linear-backoff-scheduler. It is very interesting that there is this huge difference in the performance of the yield-scheduler between these two systems. The noop-scheduler is quite fast in dedicated environments, however in multiprogrammed environments it is substantially slower than the liner-backoff- and the yield-scheduler. The two schedulers using the dynamic quanta algorithms are roughly as good as the linear-backoff-scheduler in dedicated environments. However, when four instances of the benchmark are run in parallel, the dynamic quanta schedulers perform slower than the linear-backoff scheduler and the results have a large variance.

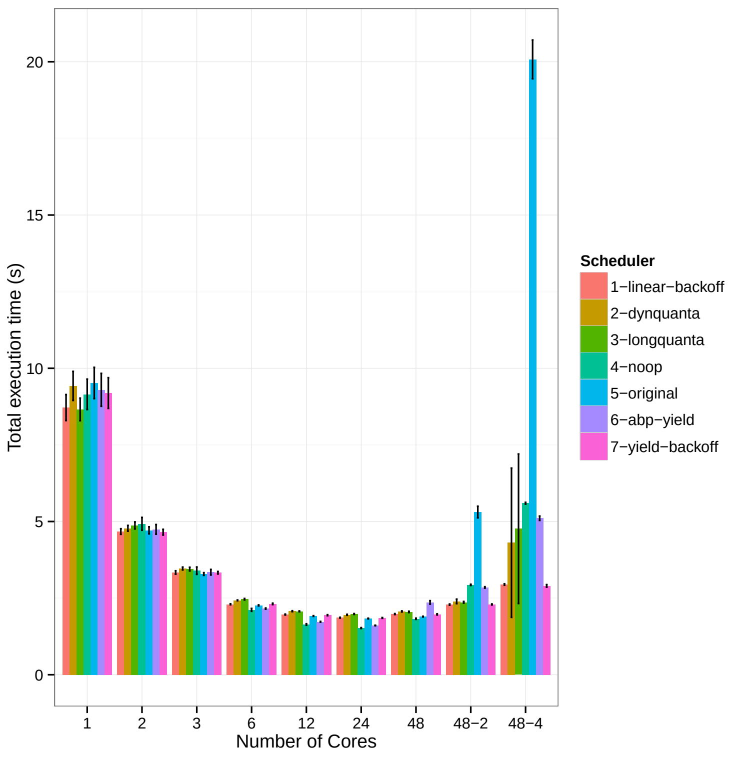


Figure 6.4: Results of LU decomposition benchmark on Saturn

6.3 Prefix Sum

Figure 6.5 shows the results for the prefix sum benchmark on Mars. On this machine all tested schedulers perform roughly the same in the dedicated environment tests. The only exception is the abp-yield-scheduler, which performs significantly worse than the other schedulers when all 80 cores are used. In multiprogrammed environments the results are not that uniform. Especially when four instances are running in parallel, more performance differences between the schedulers are visible: The original-, the longquanta- and the abp-yield-scheduler perform quite badly in this situation. Contrary the linear-backoff- and the noop-scheduler perform really well. In this benchmark run the results of some of the schedulers additionally have a quite large variance.

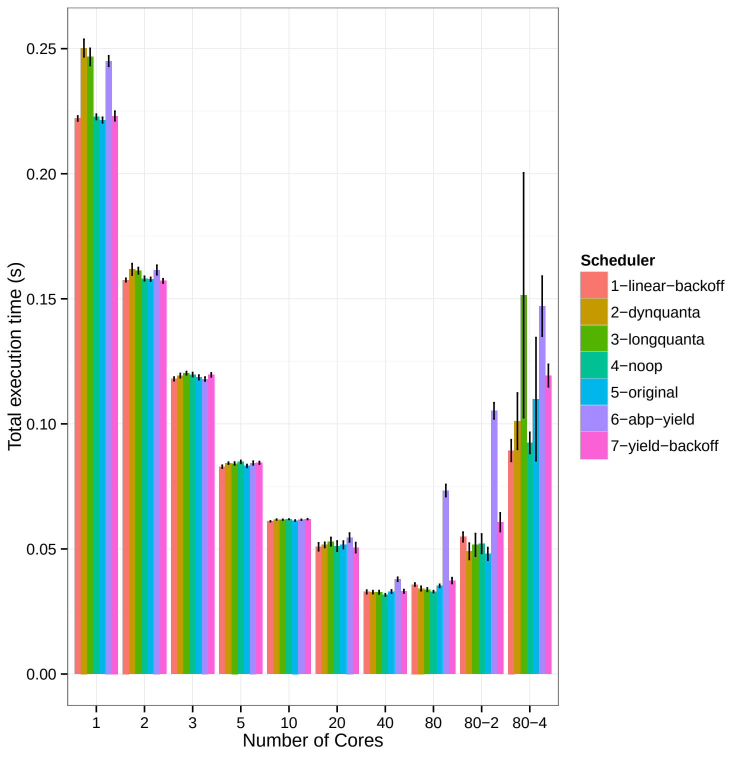


Figure 6.5: Results of Prefix Sum benchmark on Mars

Again the results on Saturn are a bit different than the results on Mars, shown by Figure 6.6. On Saturn the linear-backoff-, the yield-backoff- and the abp-yield-scheduler achieve the best performance. In dedicated environments the performance off all schedulers is roughly equally good. When two instances are run in parallel the original-scheduler is worse than the other ones. When four instances are run in parallel, additionally to the original-scheduler, the dynquanta-, longquanta- and noop-scheduler are worse than the linear-backoff- and yield-backoff-scheduler. In this case the results' variance of the dynquanta-, longquanta- and original-scheduler is again very large.

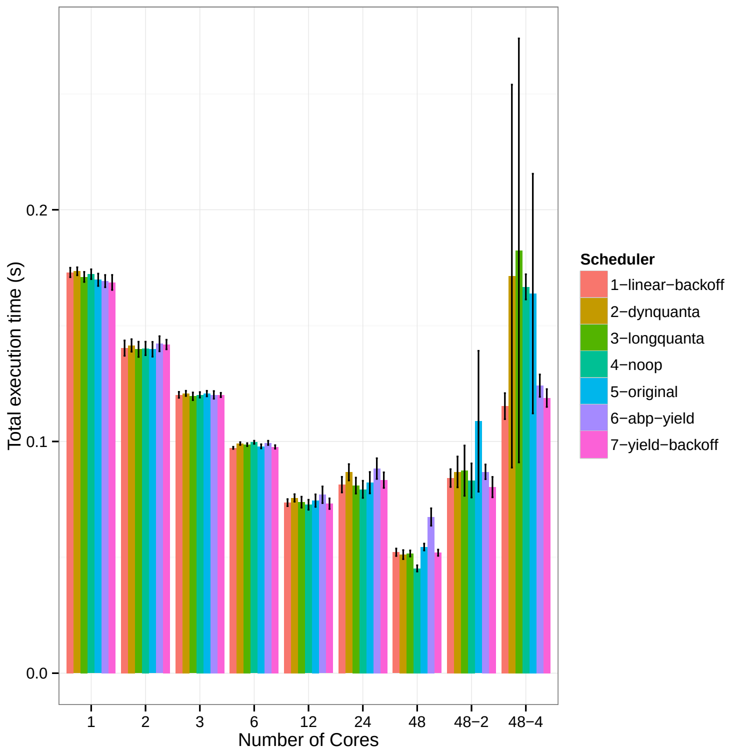


Figure 6.6: Results of Prefix Sum benchmark on Saturn

6.4 Quicksort

The results for the quicksort benchmark on Mars are shown in Figure 6.7. On Mars the original-, the yield-backoff- and the abp-yield-scheduler perform significantly worse than the other schedulers, when 40 or all 80 cores are used. When all 80 cores are used, the dynquanta- and longquanta-scheduler perform better than all other schedulers. The performance of the linear-backoff- and the original-scheduler are approximately the same.

On Saturn the results are once again quite different, which can be seen in Figure 6.8: All schedulers behave very similar, except when four instances of the benchmark are run in parallel. In this case the dynquanta-, longquanta- and original-scheduler perform significantly worse. Additionally one can see that the results of the dynquanta- and longquanta-schedulers have an extremely high variance. We repeated all tests runs of the benchmark for these schedulers, to make sure that this was not a onetime event. However, we got similar results in this retakes of the benchmark. It seems like when four instances are run in parallel they interfere quite a lot with each other and the runtime highly depends on the exact timings of these interferences.

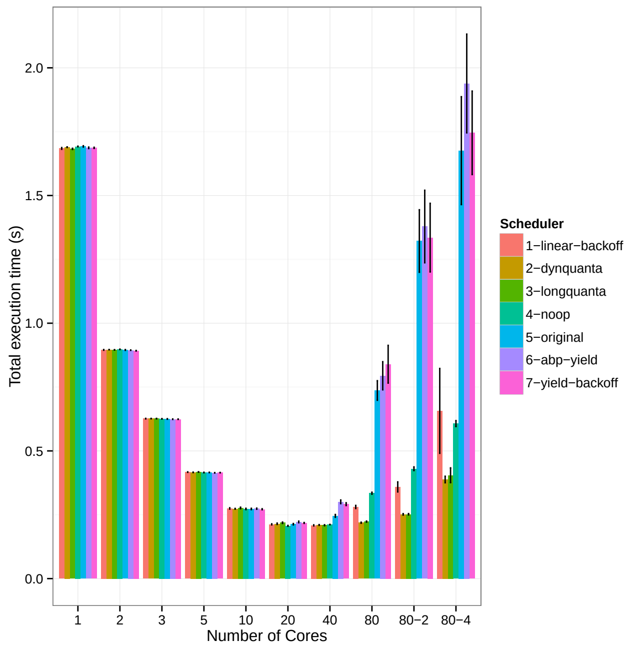


Figure 6.7: Results of Quicksort benchmark on Mars

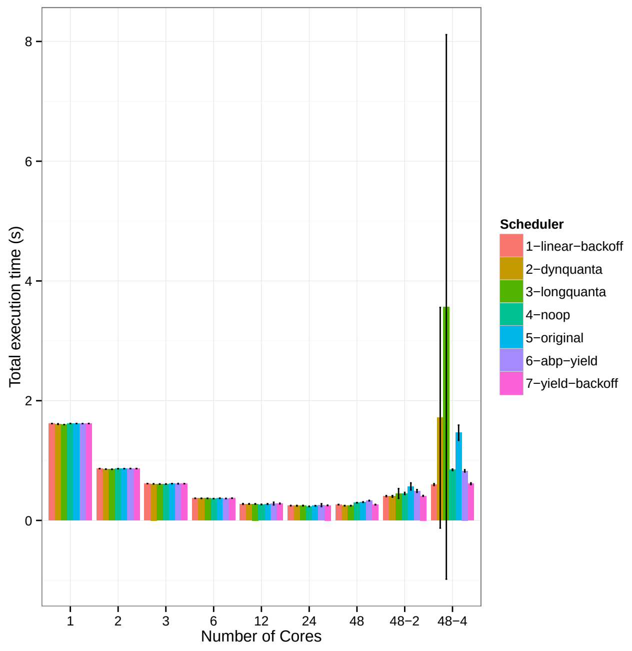


Figure 6.8: Results of Quicksort benchmark on Saturn

6.5 Single-source Shortest Path

In the single-source shortest path benchmark, which results can be seen in Figure 6.9 and Figure 6.10, the variance of the results is generally quite large, both on Mars and on Saturn.

On Mars the noop-scheduler significantly perform worse when all 80 cores are used to process the tasks. However, when four instances of the benchmark are run in parallel the noop-scheduler performs quite good and the original-scheduler performs the worst. The other schedulers more or less perform equally well.

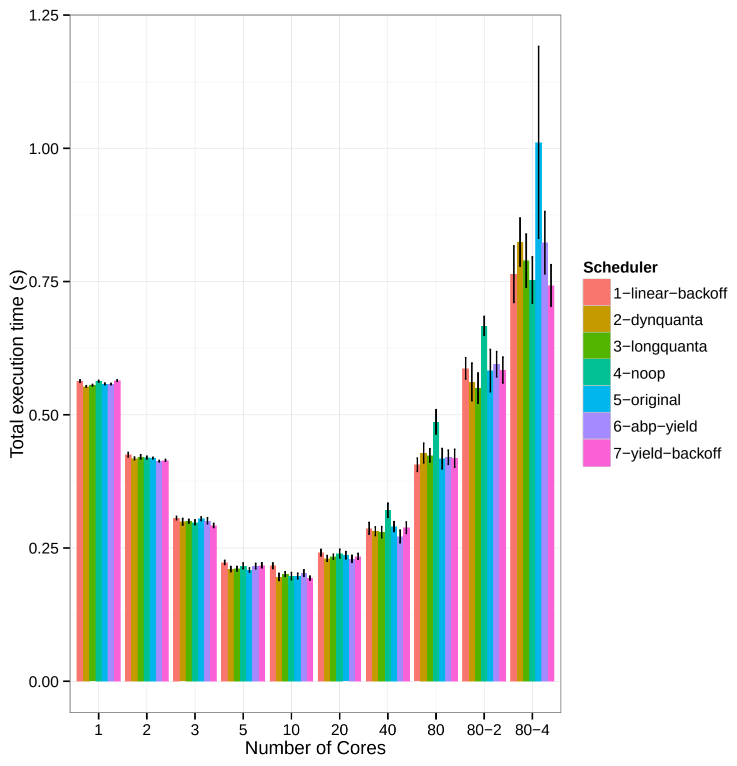


Figure 6.9: Results of Single-source Shortest Path benchmark on Mars

On Saturn the results are even harder to summarize, since they vary quite a lot. Overall the noop-scheduler performs the best, whereas the linear-backoff scheduler needs the most time to run the benchmark. This is quite contrary to the results of this benchmark on Mars.

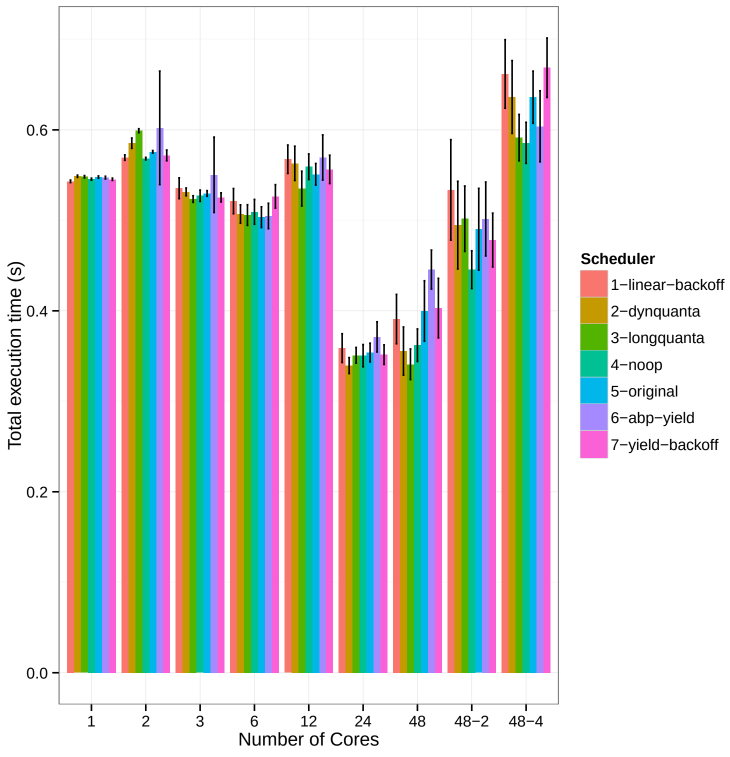


Figure 6.10: Results of Single-source Shortest Path benchmark on Saturn

6.6 Unbalanced Tree Search

As described in Section 3, three different variants of the Unbalanced Tree Search benchmark are run: The first benchmark uses a *geometric* tree (UTS-0), the second one a *hybrid* tree (UTS-4) and the last one a *binomial* tree (UTS-7).

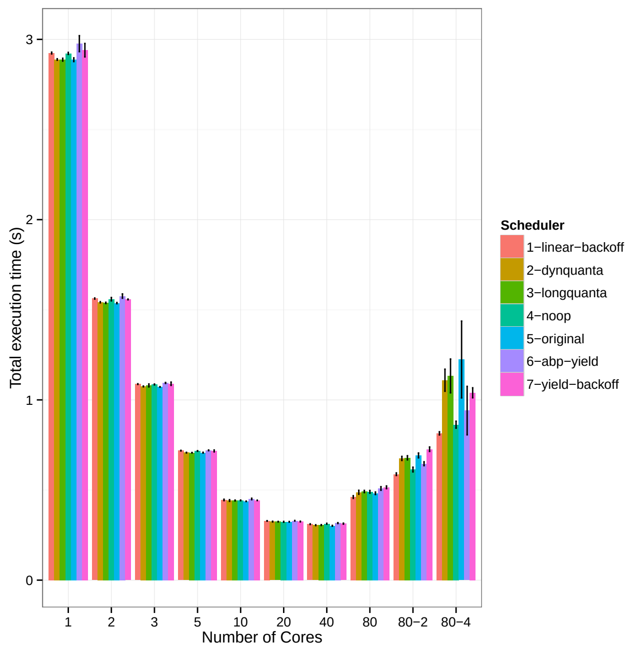


Figure 6.11: Results of Unbalanced Tree Search benchmark on Mars: geometric tree with $d = 10$, $4,130,071$ nodes (UTS-0)

Figure 6.11 shows the results of the UTS-0 benchmark on Mars. In the UTS-0 benchmark on Mars all schedulers finish the tasks in roughly the same time in dedicated environments. In multiprogrammed environments the linear-backoff- and the noop-scheduler perform better than the other ones.

On Saturn the linear-backoff-, original-, abp-yield- and the yield-backoff-scheduler perform approximately equally well. The two schedulers using the dynamic quanta algorithms perform a little bit worse than the other schedulers, and their results vary quite a lot. The results for Saturn are depicted in Figure 6.12.

In the UTS-4 benchmark (Figure 6.13) on Mars the linear-backoff-, the noop- and the abp-yield-scheduler achieve the best results.

On Saturn (Figure 6.14) the linear-backoff- and the yield-backoff-scheduler perform the best. The schedulers using the dynamic quanta algorithms and the original-scheduler have quite varying results, especially in multiprogrammed environments.

The results of the UTS-7 benchmark are depicted in Figures 6.15 and Figures 6.16. The results are very uniform, all schedulers perform the same in almost all situations, both on Mars and Saturn. Only the abp-yield-scheduler performs a little worse than the other ones. The

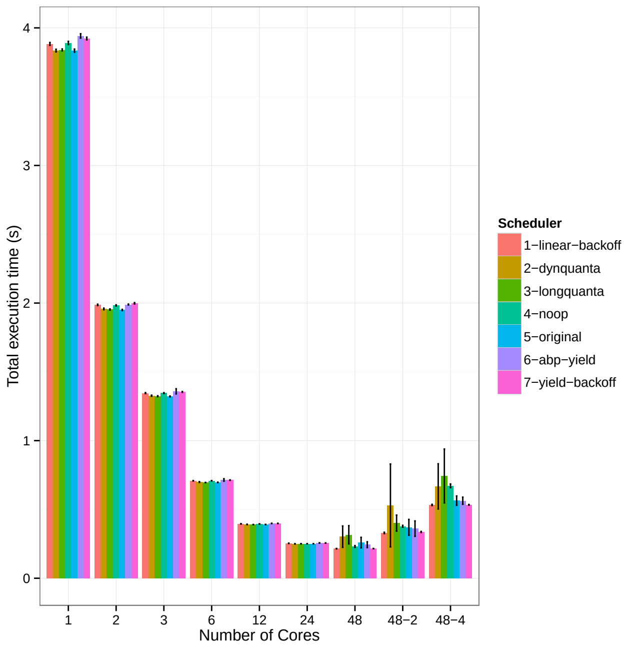


Figure 6.12: Results of Unbalanced Tree Search benchmark on Saturn: geometric tree with $d = 10, 4, 130, 071$ nodes (UTS-0)

benchmark also scales very well up to 40 cores on Mars and up to all 48 cores on Saturn. It seems like there are enough tasks to process, so that the strategy that decides when threads are blocked does not change the performance significantly.

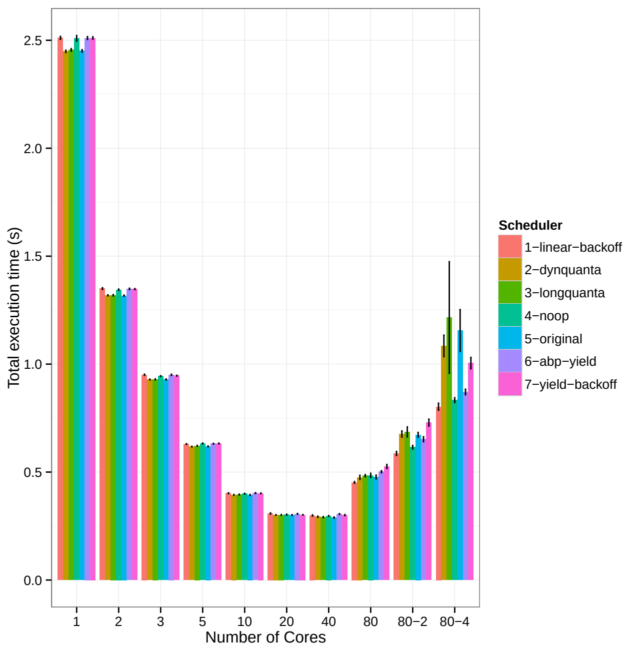


Figure 6.13: Results of Unbalanced Tree Search benchmark on Mars: hybrid tree with $d = 16$, $q = 0.234375$, $m = 4$, $4, 132, 453$ nodes (UTS-4)

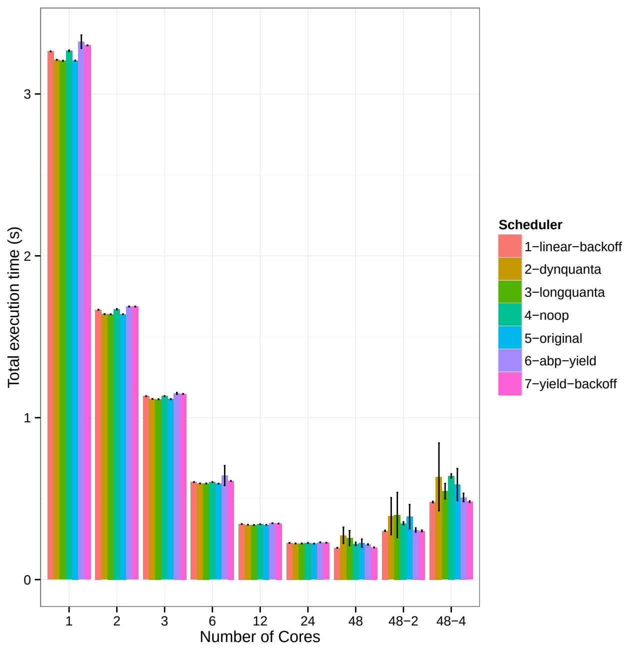


Figure 6.14: Results of Unbalanced Tree Search benchmark on Saturn: hybrid tree with $d = 16$, $q = 0.234375$, $m = 4$, $4, 132, 453$ nodes (UTS-4)

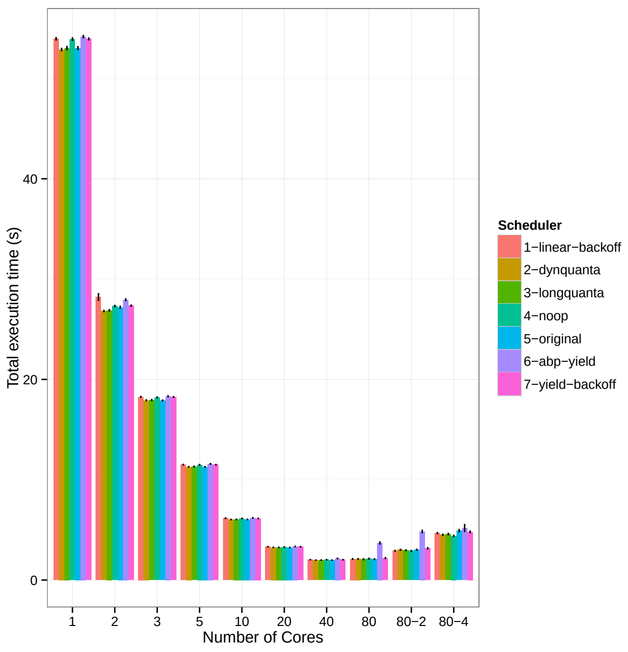


Figure 6.15: Results of Unbalanced Tree Search benchmark on Mars: binomial tree with $r = 5$, $m = 5, 111, 345, 631$ nodes (UTS-7)

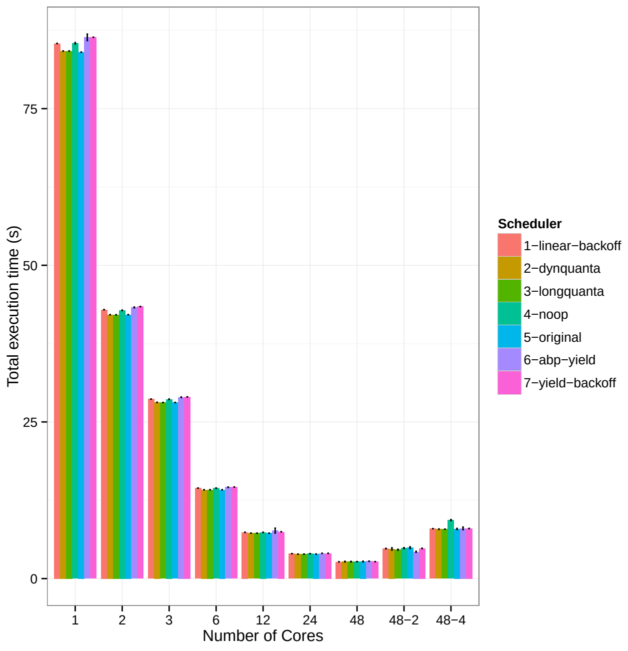


Figure 6.16: Results of Unbalanced Tree Search benchmark on Saturn: binomial tree with $r = 5$, $m = 5, 111, 345, 631$ nodes (UTS-7)

6.7 Overall

Figures 6.17 and 6.18 show the aggregated results of all benchmarks on Mars and Saturn respectively. The methodology how this summary is generated is described in Section 3.2.

The results on Mars which can be seen in Figure 6.17 are quite promising: The linear-backoff scheduler performs the best and in most cases the two dynamic quanta algorithms also perform better than the original-, abp-yield, yield-backoff- and noop-scheduler. Especially when all 80 cores are used, our newly developed schedulers clearly outperform the existing scheduler and also the yield-backoff-scheduler. Interestingly, compared with the original- and the yield-backoff-scheduler the noop-scheduler does not perform that bad at all, when all 80 cores are used.

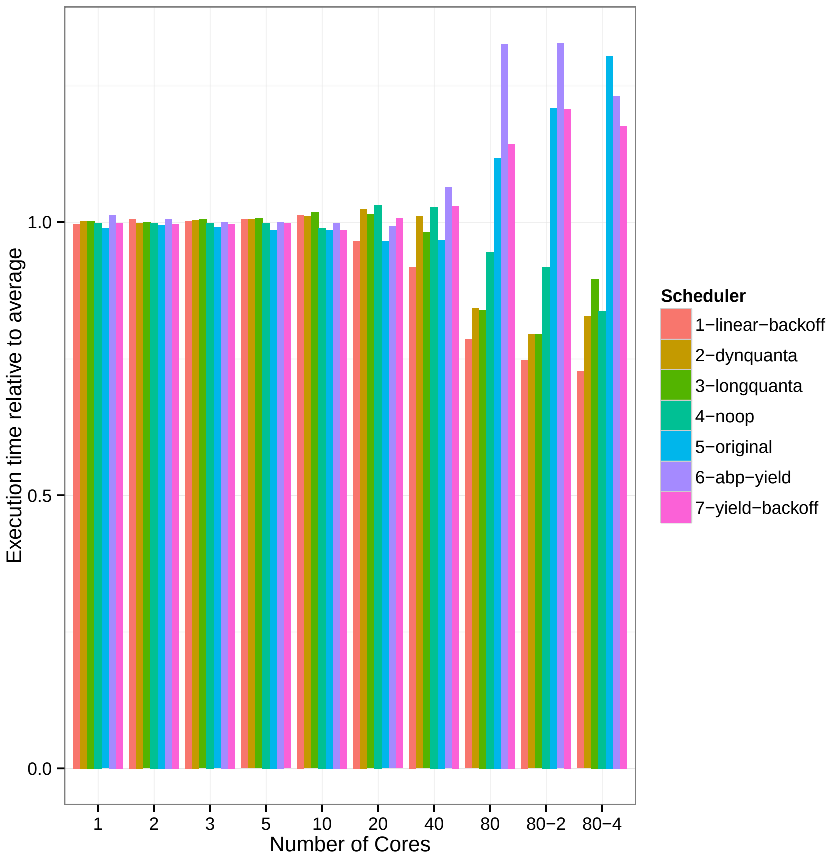


Figure 6.17: Aggregated results of benchmarks on Mars

The results on Saturn which are shown in Figure 6.18 differ from the results on Mars in some areas. First of all as long as the benchmarks are run in dedicated environments, the performance difference between the schedulers is not so big like on Mars. In the multiprogrammed environment tests the original-scheduler performs clearly worse than the better schedulers. When four instances of the benchmark are run in parallel, the longquanta-scheduler performs nearly as bad as the original-scheduler. The dynquanta- and noop-scheduler also perform significantly worse than the best two schedulers in this situation. Overall the best schedulers on Saturn are the linear-backoff- and the yield-backoff-scheduler.

The abp-yield-scheduler performs quite similar to the yield-backoff-scheduler on both systems. However, in most benchmarks the yield-backoff-scheduler performs a bit better than the abp-yield-scheduler.

An interesting observation are the performance differences of the yield-backoff-scheduler between Mars and Saturn: On Mars the yield-backoff-scheduler performs quite badly, while on Saturn together with the linear-scheduler it is the best one.

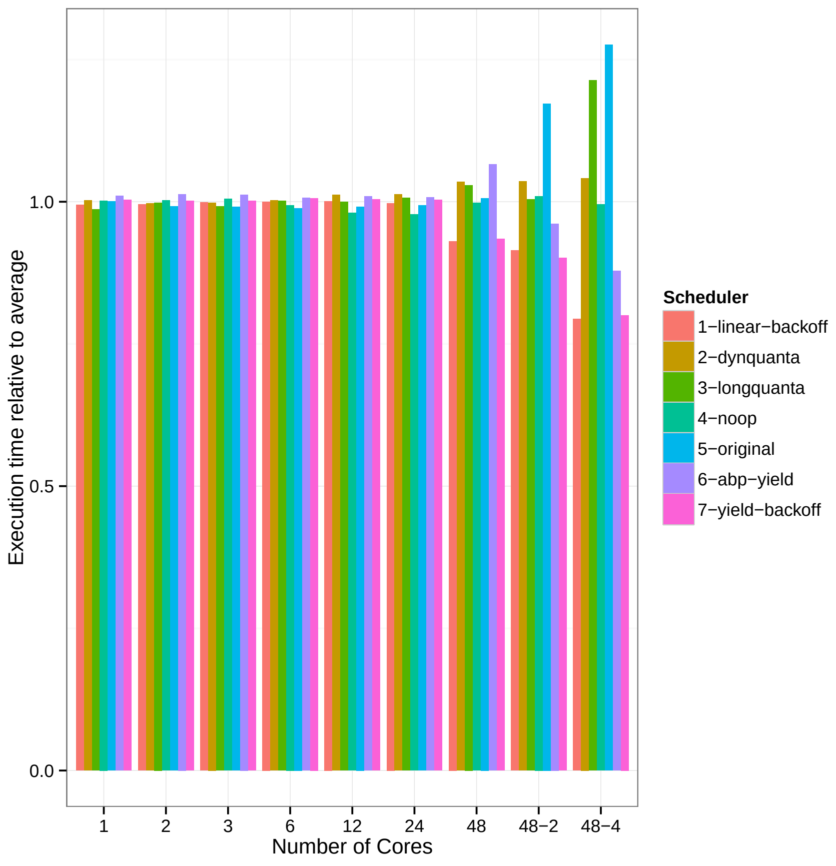


Figure 6.18: Aggregated results of benchmarks on Saturn

When analyzing all results of both systems, the scheduler that performs the best is the linear-backoff scheduler. The two schedulers using dynamic work-stealing techniques also perform better than the existing original-scheduler, but are not as good as the linear-backoff-scheduler. In comparison the basic dynamic quanta algorithm works better than the long and short quanta algorithm.

CHAPTER 7

Summary and future work

Originally the work for this thesis was focused just on adaptive work-stealing algorithms. During the work we became aware, that there exists hardly any analysis on the usage of backoffs in work-stealing schedulers. Therefore we decided to try to optimize the scheduler using a backoff as much as possible to find out what is possible with such simple means. Additionally we thought, that the performance results of our adaptive algorithms should be compared against a thoroughly optimized scheduler and not ”just an existing one”.

The work on backoffs showed, that choosing the right parameters for the backoff and trying to optimize them pays off. The scheduler using our optimized *linear backoff* is up to 18x faster than the scheduler using the original exponential backoff (LU decomposition benchmark on Mars). The work also revealed that at least in our benchmarks and on our systems the range of good parameter values for the backoffs is surprisingly large. This could mean, that the chosen parameters are suitable for a broad range of different machines. However, this assumption requires additional tests on more systems. Another interesting result is, that the type of the backoff does not matter a lot. We implemented and tested many different variants of backoffs. However, after optimizing the used parameters of them, in the end the performance of all of them was quite similar.

After the analysis of using backoffs in the work-stealing scheduler we designed a *decentralized adaptive work-stealing algorithm* that can be used on common systems without any modifications in the underlying system. Contrary to almost all existing adaptive schedulers, we decided not to use any central data structure in the scheduler. This ensures the scalability of the algorithm regardless of the number of available processors and cores. Our implementation shows that this is possible in a completely portable way, using only standard C++. Our dynamic quanta algorithm also outperforms the original scheduler that was used so far in Pheet.

In the beginning hoped that the adaptive algorithm performs better than the scheduler using a simple backoff. Much more work was put into it and it adjusts the number of used cores more dynamically. Nevertheless in the end the scheduler using the linear backoff overall performs the best. Three different explanations are possible:

- Adaptive work-stealing in general does not offer any advantage compared to a scheduler that uses a carefully tuned backoff.
- Adaptive work-stealing needs some central data structure to provide performance benefits over a non-adaptive scheduler.
- Our design or implementation of the algorithm for the decentralized adaptive work-stealing scheduler is not good enough, therefore the scheduler using a backoff performs better.

To decide which of the explanations is accurate, performance comparisons with other adaptive work-stealing schedulers would be needed. The schedulers presented in Section 2 offer performance comparisons with non-adaptive work-stealing schedulers itself. However, no comparisons with schedulers using a (optimized) linear backoff are given, which gives the best performance in our case.

To allow such comparison of different schedulers, a portable set of benchmarks would be required. Designing a set of benchmarks for task-based programming that can be used on different operating systems, in different programming languages and with different schedulers is therefore a large challenge for future work.

In retrospect this thesis contributes information to the question, which algorithm should be used to dynamically select the number of cores when processing a set of tasks with a work-stealing scheduler. Although this thesis cannot answer the question in general, it clearly shows that it is worth the effort to spend more time on this problem. In our case, which means in the Pheet framework, so far an optimized linear backoff works the best. This is an immediate improvement for all users of the framework. However, to answer the question whether this is only true for this scheduler or if completely different schedulers would also benefit from using a linear backoff in a similar way, future work in this area is needed to be done.

Bibliography

[1] Norman Abramson. The aloha system: Another alternative for computer communications. In *Proc. 1970, Fall Joint Computer Conf. (AFIPS)*, pages 281–285, New York, NY, USA, 1970. ACM.

[2] Anant Agarwal and Mathews Cherian. Adaptive backoff synchronization techniques. In *Proc. 16th Annual International Symp. on Computer Architecture (ISCA)*, pages 396–406, New York, NY, USA, 1989. ACM.

[3] Kunal Agrawal, Yuxiong He, Wen-Jing Hsu, and Charles E. Leiserson. Adaptive Scheduling with Parallelism Feedback. In *Proc. 11th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPOPP)*, pages 100–109, 2006.

[4] Kunal Agrawal, Yuxiong He, and Charles E. Leiserson. Adaptive Work Stealing with Parallelism Feedback. In *Proc. 12th ACM SIGPLAN Symp. Principles and Practice of Parallel Programming (PPOPP)*, pages 112–120, 2007.

[5] David Aldous. Ultimate instability of exponential back-off protocol for acknowledgment-based transmission control of random access communication channels. *IEEE Transactions on Information Theory*, 33(2):219–223, Mar 1987.

[6] Andrei Alexandrescu. *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley, 2001.

[7] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, S Hammerling, Alan McKenney, et al. *LAPACK Users' guide*, volume 9. Siam, 1999.

[8] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for Multiprogrammed Multiprocessors. In *Proc. 12th ACM Symp. Parallel Algorithms and Architectures (SPAA)*, pages 119–129, 1998.

[9] Petra Berenbrink, Tom Friedetzky, and Leslie Ann Goldberg. The natural work-stealing algorithm is stable. *SIAM Journal on Computing*, 32(5):1260–1279, 2003.

[10] Giuseppe Bianchi. Performance analysis of the ieee 802.11 distributed coordination function. *IEEE Journal on Selected Areas in Communications*, 18(3):535–547, March 2000.

[11] Guy E Blelloch. Scans as primitive parallel operations. *IEEE Transactions on Computers*, 38(11):1526–1538, 1989.

[12] Robert D. Blumofe and Charles E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proc. 35th IEEE Symp. Foundations of Computer Science (FOCS)*, pages 356–368, 1994.

[13] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 180–186, Feb 2010.

[14] F. Warren Burton and M. Ronan Sleep. Executing functional programs on a virtual tree of processors. In *Proc. 1981 Conference on Functional Programming Languages and Computer Architecture (FPCA)*, pages 187–194. ACM, 1981.

[15] Yangjie Cao, Hongyang Sun, Depei Qian, and Weiguo Wu. Stable Adaptive Work-Stealing for Concurrent Multi-core Runtime Systems. In *Proc. 13th IEEE Int. Conf. High Performance Computing and Communications (HPCC)*, 2011.

[16] Quan Chen, Long Zheng, and Minyi Guo. Dws: Demand-aware work-stealing in multi-programmed multi-core architectures. In *Proc. of Programming Models and Applications on Multicores and Manycores*, page 131. ACM, 2014.

[17] Teodor Gabriel Crainic, Bertrand Le Cun, and Catherine Roucairol. Parallel branch-and-bound algorithms. *Parallel Combinatorial Optimization*, pages 1–28, 2006.

[18] Andreas Crauser, Kurt Mehlhorn, Ulrich Meyer, and Peter Sanders. A parallelization of dijkstra's shortest path algorithm. In *Mathematical Foundations of Computer Science 1998*, pages 722–731. Springer, 1998.

[19] Xiaoning Ding, Kaibo Wang, Phillip B Gibbons, and Xiaodong Zhang. BWS: Balanced Work Stealing for Time-Sharing Multicores. In *Proc. 7th ACM European Conf. Computer Systems (EuroSys)*, pages 365–378, 2012.

[20] Jeff Edmonds. Scheduling in the Dark. *Theoretical Computer Science*, 235(1):109–141, 2000.

[21] Paul Erdős and Alfréd Rényi. On random graphs. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[22] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Proc. 1998 ACM SIGPLAN Conf. Program Language Design and Implementation (PLDI)*, pages 212–223, 1998.

[23] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[24] Robert H. Halstead, Jr. Implementation of multilisp: Lisp on a multiprocessor. In *Proc. 1984 ACM Symp. on LISP and Functional Programming (LFP)*, pages 9–17, New York, NY, USA, 1984. ACM.

[25] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Elsevier, 2012.

[26] Alexey Kukanov and Michael J. Voss. The foundations for scalable multi-core software in intel threading building blocks. *Intel Technology Journal*, 11(4), 2007.

[27] Byung-Jae Kwak, Nah-Oak Song, and Leonard E. Miller. Performance analysis of exponential backoff. *Networking, IEEE/ACM Transactions on*, 13(2):343–355, April 2005.

[28] Cheng-Han Lin, Ce-Kuen Shieh, Wen-Shyang Hwang, and Chih-Heng Ke. An exponential-linear backoff algorithm for contention-based wireless networks. In *Proc. International Conference on Mobile Technology, Applications, and Systems (Mobility)*, pages 42:1–42:6, New York, NY, USA, 2008. ACM.

[29] Cathy McCann, Raj Vaswani, and John Zahorjan. A Dynamic Processor Allocation Policy for Multiprogrammed Shared-Memory Multiprocessors. *ACM Transactions Computer Systems (TOCS)*, 11(2):146–178, 1993.

[30] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, February 1991.

[31] Ulrich Meyer and Peter Sanders. Δ-stepping: a parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1):114–152, 2003.

[32] Stephen Olivier, Jun Huan, Jinze Liu, Jan Prins, James Dinan, P Sadayappan, and Chau-Wen Tseng. Uts: An unbalanced tree search benchmark. In *Languages and Compilers for Parallel Computing*, pages 235–250. Springer, 2007.

[33] Artur Podobas, Mats Brorsson, and Karl-Filip Faxén. A comparison of some recent task-based parallel programming models. In *Proc. 3rd Workshop on Programmability Issues for Multi-Core Computers*, 2010.

[34] Gregorio Quintana-Ortí, Enrique S. Quintana-Ortí, Robert A. Van De Geijn, Field G. Van Zee, and Ernie Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. *ACM Trans. Math. Softw.*, 36(3):14:1–14:26, July 2009.

[35] Hongyang Sun and Wen-Jing Hsu. Adaptive B-Greedy (ABG): A Simple yet Efficient Scheduling Algorithm. In *Proc. 22nd IEEE Int. Symp. Parallel and Distributed Processing (IPDPS)*, pages 1–8, 2008.

[36] Herb Sutter. The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software. *Dr. Dobb's Journal*, 30(3):202–210, 2005.

[37] Philippas Tsigas and Yi Zhang. A simple, fast parallel implementation of quicksort and its performance evaluation on sun enterprise 10000. In *Proc. 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 372–381. IEEE, 2003.

[38] Georgios Varisteas and Mats Brorsson. Automatic Adaptation of Resources to Workload Requirements in Nested Fork-join Programming Models. Technical Report 12:04, KTH, Software and Computer systems, SCS, 2012.

[39] Georgios Varisteas and Mats Brorsson. Palirria: Accurate On-line Parallelism Estimation for Adaptive Work-Stealing. In *Proc. 2014 ACM PPOPP Int. Workshop Programming Models and Applications for Multicores and Manycores (PMAM)*, pages 120–131, 2014.

[40] Martin Wimmer. *Variations on Task Scheduling for Shared Memory Systems*. PhD thesis, Vienna University of Technology, 2014.

[41] Martin Wimmer and Jesper Larsson Träff. Work-stealing for mixed-mode parallelism by deterministic team-building. In *Proc. 23rd Annual ACM Symp. on Parallelism in Algorithms and Architectures (SPAA)*, pages 105–116, New York, NY, USA, 2011. ACM.

[42] Martin Wimmer and Jesper Larsson Träff. A work-stealing framework for mixed-mode parallel applications. In *Workshop on Multi-threaded Architectures and Applications (MTAAP)*, 25th IEEE International Symp. on Parallel and Distributed Processing (IPDPS), 2011.