Dynamic Load Balancing on Massively Parallel Computer Architectures

B A C H E L O R A R B E I T

zur Erlangung des akademischen Grades

Bachelor of Science (B.Sc.)
im Fach Informatik

eingereicht am Institut für
Informatik, Fachbereich Mathematik und Informatik
Freie Universität Berlin

von
Florian Wende
geboren am 06.05.1985 in Dresden

Präsident der Freie Universität Berlin:
Prof. Dr. Peter-André Alt

Dekan des Fachbereich Mathematik und Informatik:
Prof. Dr. R. Klein

Gutachter:
1. Prof. Dr. H. Alt
2. Prof. Dr. A. Reinefeld

eingereicht am: 12.04.2013
Tag der mündlichen Prüfung:

Abstract

This thesis reports on using dynamic load balancing methods on massively parallel computers in the context of multithreaded computations. In particular we investigate the applicability of a randomized work stealing algorithm to ray tracing and breadth-first search as representatives of real-world applications with dynamic work creation. For our considerations we made use of current massively parallel hardware accelerators: Nvidia Tesla M2090, and Intel Xeon Phi. For both of the two we demonstrate the suitability of the work stealing scheme for the said real-world applications. Also the necessity of dynamic load balancing for irregular computations on such hardware is illustrated.

Keywords:
GPGPU, CUDA, MIC, Xeon Phi, Dynamic Load Balancing, Work Stealing, Ray Tracing, Breadth-First Search

Zusammenfassung

Vorliegende Bachelorarbeit befasst sich mit Methoden der dynamischen Lastbalancierung auf massiv parallelen Computern im Rahmen von mehrprozess gestützten Ausführungen von Programmen. Im einzelnen wird die Eignung eines randomisierten Work-Stealing Algorithmus für die Ausführung realer Anwendungen mit dynamischer Arbeitserzeugung, wie Ray-Tracing und Breitensuche, untersucht. Für die entsprechenden Betrachtungen werden aktuelle massiv parallele Hardwarebeschleuniger vom Typ Nvidia Tesla M2090 und Intel Xeon Phi verwendet. Für beide Beschleunigertypen konnte die Tauglichkeit des Work-Stealing Schemas für die genannten Anwendungen gezeigt werden. Ebenfalls wird die Notwendigkeit der Verwendung dynamischer Lastausgleichsmethoden für irregulare Berechnungen auf der genannten Hardware verdeutlicht.

Schlagwörter:
GPGPU, CUDA, MIC, Xeon Phi, Dynamische Lastbalancierung, Work-Stealing, Ray-Tracing, Breitensuche

Contents

1. Introduction

The development of parallel computer systems and concepts of parallel programming both have their origin in the mid-1960s. Primarily driven by the strongly increasing demand for computational power in almost all kinds of scientific fields of research, single processor systems rapidly evolved to those containing multiple processors. Today we are faced with multi-core CPUs in standard consumer products, high-performance computer systems in data centers, and vector processors and hardware accelerators in special purpose computers. Currently, the use of massively parallel hardware accelerators drive the raw compute performance of the fastest systems towards several PetaFLOPS[1] and beyond.

For about 5 years, programmers from different fields of research report about significant performance gains when executing adapted programs on GPUs (GPU—Graphics Processing Unit) compared to executing them on a standard multi-core CPU. While GPUs are well suited for data parallel applications with known launch bounds, dynamic task parallelism is a discipline even current GPUs seem to be at a disadvantage to multi-core CPUs [TLO12]. The point why GPUs seem to be less suitable for such kind of applications is irregularities in program control flows and task execution times, and task (re)distribution at runtime when work is created dynamically. According to Burtscher et al. [BNP12], irregularities in control flow and memory access patterns do not necessarily result in bad performance of GPU programs. For almost all application domains considered by the authors—graph theory, satisfiability theory, computational geometry, to name a few—they found current GPUs be less sensitive to unfortunate memory access patterns due to the introduction of data caches on the latest models. The more interesting issue thus is dynamic task parallelism, which is more or less in contrast with the functioning of the GPU hardware.

Similar issues may also apply to hardware accelerators other than GPUs, and in particular to heterogeneous computer systems made up of different kinds of processors.

[1] FLOPS—Floating Point Operations Per Second. $1\,\text{PetaFLOPS} = 1 \times 10^{15}\,\text{FLOPS}$.

1.1. Related Work

With respect to dynamic work creation, there is a broad class of algorithms that actually develop their parallelism just at runtime—divide and conquer algorithms, for instance. For such algorithms to execute on massively parallel computers, a mechanisms is needed that (re)distributes newly created work to processors that are non-busy at that time. Investigations on applying load balancing schemes, known to serve well on traditional multiprocessor systems, to the GPU can be found in [CT08, CGSS11, TLO12], for instance. While Cederman et al. [CT08] and Chatterjee et al. [CGSS11] obtain good results with a work stealing based load balancing scheme (the authors do not handle dependencies between tasks), Tzeng et al. [TLO12] have their focus on irregular workloads with dependencies. Unfortunately, the task parallel programming model for the GPU proposed by the authors was inferior to its CPU counterpart.

Besides using load balancing schemes on a particular kind of multiprocessor, distributing work amongst the components of a heterogeneous system is another challenging application domain of load balancing schemes. In this field several investigations have been done. Well known outcomes here are the StarPU [ATNW11] and the X-Kaapi [GLFR12] programming APIs. Both of the two aim for making the heterogeneous computer's compute units—CPU cores, GPUs, and maybe other hardware accelerators—execute a multithreaded computation in cooperative manner. For that purpose X-Kaapi implements a work stealing scheduler, whereas StarPU allows to dynamically switch between different scheduling schemes including work stealing. The key difference to using load balancing on the hardware accelerator itself is the abstraction of the underlying hardware. X-Kaapi and StarPU abstract accelerators like CPU cores. Large portions of the computation are scheduled to the accelerator in hope of fast execution (offload computation), while small portions are assigned to the CPU core(s). Load balancing on the accelerators is not addressed by these APIs (to our knowledge).

Other examples of using hardware accelerators for offload computations are MAGMA, and TBLAS [SYD09, SHT+12]. Both libraries focus on solving dense linear algebra problems. The approach is to replace BLAS (Basic Linear Algebra Subroutine) routines by so-called task-based linear algebra subroutines that are dynamically created during the execution of the computation, and then are distributed to available processors. The scheduling mechanism has to respect dependencies between the tasks.

At the core of all these APIs and libraries is the integration of massively parallel hardware accelerators. As vendors of such accelerators are going to provide their devices with increasingly more and more processors, further investigations on load balancing schemes seem to be crucial to make a wide range of applications, including those with irregular workloads, benefit from the massive compute performance of the hardware.

1.2. Contribution of the Thesis

In this thesis we extend the aforementioned investigations on load balancing using a current Nvidia GPU and Intel's Xeon Phi hardware accelerator. We want to emphasize that our focus will be on load balancing on the hardware accelerator itself. A combination with StarPU or X-Kaapi might be the starting point of further investigations. Unlike in [CT08] and [CGSS11], we also compare the performance of our implementations against equivalent implementations on a standard x86 multi-core CPU—the functioning of the load balancing schemes on the GPU and the Xeon Phi does not mean that the overall performance of a respective application is superior to its CPU version, as experienced for applications with no irregular workloads. We mimic the task parallel Cilk programming model—implementing work stealing on x86 CPUs and the Xeon Phi—on both GPU and Xeon Phi, and evaluate its performance by directly comparing against Cilk (we also address dependencies between tasks). A downgraded version of this load balancing scheme will be applied to real-world problems, ray tracing and breadth-first search, namely.

Chapter 2 is concerned with multithreaded computations and dynamic load balancing. After briefly introducing the graph theoretical model of multithreaded computations, the focus will be on scheduling aspects. We will draw on the work of D. Cederman and P. Tsigas [CT08], and R. D. Blumofe and Ch. E. Leiserson [BL93, BL99].

In Chapter 3 we introduce the massively parallel computer hardware used in this thesis—Nvidia Fermi GPU, and Intel Xeon Phi, namely. We then are concerned with the implementation and the evaluation of concurrent data structures—'lock' and 'dequeue'. Further, we give an implementation of the work stealing load balancing scheme that puts the graph theoretical model of multithreaded computations into practice.

Chapter 4 applies the work stealing scheme to real-world applications, ray tracing and breadth-first search (BFS). We evaluate the performance of the ray tracer using a static work distribution scheme, a centralized work pool, and work stealing. For the BFS algorithm, we restrict our considerations to the Xeon Phi and the CPU.

Chapter 5 summarizes the results of this thesis. We also list some interesting issues that might serve as a basis for further investigations.

2. Dynamic Task Parallelism in Multithreaded Computations

This chapter is concerned with scheduling schemes in the context of multithreaded computations. After having briefly introduced the notion of a multithreaded computation and its graph theoretical representation, we list common approaches to handle such computations. In particular, we amplify greedy schedules, and introduce a randomized work stealing scheme. Throughout this chapter, we follow the work of D. Cederman and P. Tsigas [CT08], and R. D. Blumofe and Ch. E. Leiserson [BL93, BL99].

2.1. Multithreaded Computations

A *multithreaded computation* is a composition of *threads*, which are sequential orderings of tasks each. A *task* here refers to an abstract operation which in the context of the computation is atomic, that is, for the multithreaded computation it is not split up into subtasks. Tasks t_j within a thread $\mathcal{T}$ are executed in a predefined order, with tasks that are direct successors being connected by *continue edges* going from a task t_j to its direct successor $succ(t_j)$. Threads may create new threads during their execution, called *child threads*. The thread which *spawns* the child thread acts as the *parent* of the child, and as an ancestor of the child's child(ren). In this way, threads are organized into an *activation tree*, with parent threads and child threads connected by *spawn edges* going from the parent to the child thread. If there is some kind of producer-consumer relation between threads, say, thread $\mathcal{T}_0$ is the consumer and threads $\mathcal{T}_{i:1\leq i\leq n}$ are the producers, then there are n *data-dependency edges* going from any task t_{j_i} in $\mathcal{T}_{i:1\leq i\leq n}$ to a somehow distinguished task t_* in $\mathcal{T}_0$, where t_* consumes the data. A multithreaded computation can be represented by a directed graph with its nodes corresponding to tasks, and with the different types of edges introduced above (see Fig. 2.1).

An *execution schedule* for a multithreaded computation is a task-processor mapping which at each step of the schedule assigns a task to each processor of a parallel computer for execution. The schedule depends on the multithreaded computation itself, and on the number of processors of the parallel computer used for the execution. At every step of the schedule each processor is assigned at most one task. For an execution schedule

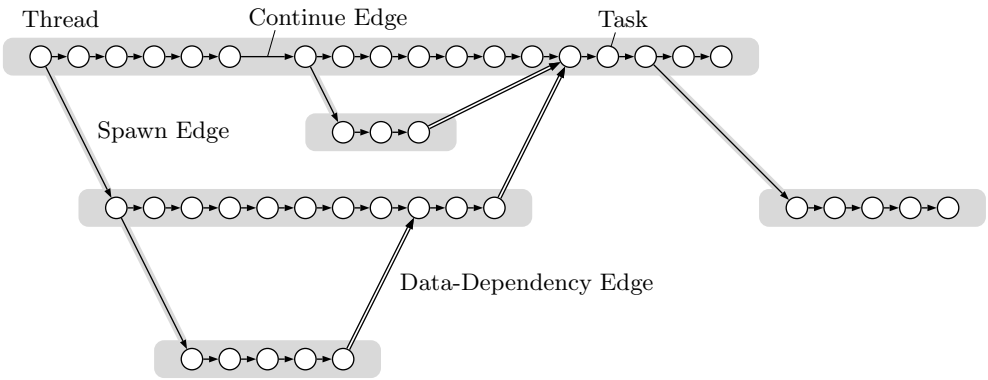

Figure 2.1.: Representation of a multithreaded computation by means of a directed graph. Threads are denoted by gray shaded boxes, and tasks correspond to open circles. Threads are connected by spawn edges and data-dependency edges (if there are any), whereas successive tasks within a thread are connected by continue edges. The image is based on similar drawing in [BL99].

to be valid, ordering constraints given by continue and data-dependency edges must be preserved, that is, tasks can be executed if and only if all its predecessor tasks, connected to it via continue or data-dependency edges, have been executed. The execution of the consuming thread thus cannot continue until the respective data dependencies (if there are any) are resolved—in the meantime the consuming thread *stalls*.

For subsequent considerations we assume that there is a valid execution schedule for the multithreaded computation of interest. Then the multithreaded computation can be represented by a directed acyclic graph (DAG).[1] We further assume that for each thread the number of data-dependency edges incident on it is bounded above, so that the computation can be performed within a finite amount of time, and with a finite amount of computer resources.

2.2. Dynamic Load Balancing

A general multithreaded computation consists of a certain number of threads with some of them created (spawned) by other threads during their execution (see Fig. 2.1). Since spawning threads from within a thread is somehow similar to a subroutine call, except that the spawned thread can be executed concurrently to the thread that did the spawn, a scheme is needed that presents one of the two threads to processors that are currently non-busy, and allows any of them to start executing this thread. In this way the amount of parallelism can be dynamically adapted to the resources provided by the computer system. This point is addressed by *dynamic load balancing schemes*.

[1] If the graph would not be acyclic, the presence of cycles results in the computation will stall at any step due to data dependencies that cannot be resolved.

At the core of load balancing in the context of multithreaded computations are concurrent data structures that on the one hand allow several competing processors each to access the data atomically, that is, without being interrupted by other processors, and on the other hand assure high throughput and high availability. To meet these requirements, several popular load balancing schemes draw on atomic hardware primitives such as compare-and-swap and test-and-set. Whether these primitives are available or not depends on the hardware. However, all these load balancing schemes have in common that a shared data object—queue, dequeue, stack, or heap, for instance—is used to store all threads that are created before and during the execution of the application. When the data structure is 'not empty' processors that are ready for work repeatedly try to get a thread from it and start executing the respective thread's tasks until the computation is done. Processors which spawn threads 'add' them to the data structure to enable other processors to acquire these threads and so to leave their non-busy state.

At the core of accessing a shared data structure is *synchronization* in order to assure atomicity in the sense described above. Synchronization schemes can be divided into the following categories [CT08, TZ00]:

Blocking: Shared data objects are altered only within *critical sections* which are surrounded by a pair of 'locking' and 'unlocking' functions, traditionally based on mutexes or semaphores ('lock' for simplicity). A processor that enters the critical section is said to hold the lock associated with the data structure and the critical section, respectively. If the processor leaves the critical section the lock is released and other processors may acquire it. Processors that try to enter the critical section wait for the event 'lock was released and can be acquired' to occur. In this way all alterations with respect to the shared object are done mutual exclusive.
Unfortunately, blocking access to shared objects suffers from *contention* (multiple processors repeatedly check if a lock has been released or not, causing expensive busy-waiting), *convoying* (the processor holding the lock is preempted and thus is unable to release the lock until it is brought back for execution, causing other processors to wait), and from the risk of *deadlocks* (the processor holding the lock crashes and the shared object is locked permanently, or two or more processors circularly wait for locks held by other processors).

Non-blocking (Optimistic Concurrency Control): Shared data objects can be accessed by multiple processors at the same time without enforcing mutual exclusion. The risk of deadlocks thus is eliminated. Atomicity with respect to altering the shared object in a consistent way is guaranteed by applying changes to the object using atomic hardware primitives only—just one processor, out of a set of concurrent processors accessing the data object, can finish its operation(s) successfully such

that the state of the object before and at the end of the operation is identical, that is, none of the other processors has altered the object in the meantime. All processors that failed altering the data object successfully may have a further try if necessary.

Lock-free: As multiple processors concurrently alter a shared data object, with just one of them achieving success while the other ones may repeatedly try again, there is the possibility for individual processors to starve, that is, some processors will never be successful. However, system-wide progress is guaranteed.

Wait-free: In addition to lock-freedom, wait-freedom guarantees for each processor to complete its altering of the shared object within a bounded number of steps. Wait-freedom thus guarantees starvation-freedom.

Which kind of synchronization scheme to use depends on the application and on the particular computer system used. While for massively parallel computer systems lock-free (and wait-free) synchronization mechanisms are the matter of choice, due to almost linear scaling with number of processors [TZ00], lock-based synchronization schemes might be suitable for systems with just a few processors.

Although lock- and wait-free synchronization methods are superior to blocking ones, they are usually harder to design.

2.2.1. Dynamic Load Balancing Methods

In this subsection we summarize some load balancing methods described in [CT08]. Other than the authors we do not explicitly distinguish between thread list, thread queue, and thread dequeue, since concrete implementations are usually array-based, and 'list', 'queue', 'dequeue', etc. then are just different prescriptions of which elements of the array can be accessed next. We therefore want to refer to these data structures as *thread pool*s hereafter.[2]

In the context of multithreaded computations, we distinguish between using a single centralized thread pool or multiple distributed thread pools, processors can get threads from. Further we may distinguish between thread pools that use blocking and those using non-blocking synchronization mechanisms. Also the question of whether our thread pool(s) should allow adding newly created threads at runtime or not might be reasonable.

[2] In [CT08] the meaning of 'thread' and 'task' is different from what we have used so far. In the notation of the authors tasks map onto threads, and threads executing tasks map onto processors executing threads.

With respect to these issues, the following combinations are conceivable:

Blocking centralized thread pool: A simple thread pool protected by a lock. Since only one processor can access the pool at any time, there is huge potential for contention and convoying. If processors frequently access the pool, the available parallelism is limited by the thread pool itself, rendering it unsuitable for massively parallel computer systems. Threads created at runtime can be easily added to the pool unless its capacity is reached.

Non-blocking centralized thread pool: A pool which instead of a lock uses atomic hardware primitives to assure mutual exclusion. Although there is no contention, for a large number of processors trying to access the pool, the delay, due to just one of the competing processors successfully removes or adds a thread at any time, may become extremely large. Thus, the actual available parallelism is again limited by the thread pool itself when used for massively parallel computers.

Blocking distributed thread pools: Compared to the blocking centralized thread pool, the available parallelism should be measurably larger as processors may access different pools when trying to acquire new threads. For each processor selecting a pool for access can be done, for instance, in round robin fashion or at random.

Non-blocking distributed thread pools: Similar to the non-blocking centralized thread pool, but with processors selecting pools for access, for instance, in round robin fashion or at random.

For (massively) parallel computers the latter approach is the most promising. In fact, it is the data structure used for the work stealing algorithm, with each processor being the owner of a non-blocking (lock-free) thread pool. The owner can add and remove threads from its pool, whereas all other processors are just allowed to remove (or *steal*) threads, hence the name *work stealing*. Another scheduling paradigm which addresses scheduling multithreaded computations is *work sharing*, where dynamically created threads are migrated to other processors in hopes of distributing the work to underutilized processors. The work stealing scheme will be described in Sec. 2.4. Beforehand, we introduce some useful notation and amplify the greedy-scheduling theorem.

2.3. Greedy-Scheduling of Multithreaded Computations

Greedy schedules are those in which at each step of the execution $p \leq P$ tasks execute on up to P processors if at the respective step p tasks are ready for execution. To get a deeper insight into greedy schedules of multithreaded computations, we need to introduce some notation. We again follow R. D. Blumofe and Ch. E. Leiserson [BL99, BL93].

According to Sec. 2.1, a multithreaded computation can be represented by a bounded-edge DAG, with nodes corresponding to tasks, and tasks being connected by continue, spawn, and data-dependency edges. Tasks that are connected by continue edges belong to the same thread, and the multithreaded computation consists of a finite number of threads with some of them created at runtime by other threads.

Since general multithreaded computations with arbitrary data dependencies cannot be scheduled efficiently, subsequent considerations are for the subclass of strict and fully-strict multithreaded computations. A multithreaded computation is said to be *strict* if for all threads all data-dependency edges from a thread go to an ancestor of the thread in the activation tree. It is said to be *fully-strict* if for all threads all data-dependency edges from a thread go to its parent thread. Strict and fully-strict computations correspond to those that can be executed in a depth-first manner on a single processor system.

To quantify the space and time bounds for an execution schedule $\mathcal{X}$ of a multithreaded computation, we assume a computer system with P processors. For each thread $\mathcal{T}_i$ that is executed, an activation frame $\mathcal{F}_i$ is allocated, and the thread is said to be *alive*. $|\mathcal{F}_i|$ refers to the amount of memory used to store all data needed for the execution of thread $\mathcal{T}_i$. The activation frame $\mathcal{F}_i$ of a thread $\mathcal{T}_i$ is deallocated if and only if $\mathcal{T}_i$ is done and all its children's activation frames have been deallocated. The thread then *dies*. At a given step x of the execution schedule $\mathcal{X}$, the portion of the activation tree $\mathcal{A}$ consisting of those threads that are alive at that step defines the *activation subtree* $\bar{\mathcal{A}}(x)$ at step x. The space $S(x)$ required for the execution schedule at step x thus is the size of all activation frames used by all threads in the activation subtree $\bar{\mathcal{A}}(x)$, that is, $S(x) = \sum_{i:\mathcal{T}_i \in \bar{\mathcal{A}}(x)} |\mathcal{F}_i|$. The space requirement $S_P(\mathcal{X})$ for a P-processor execution schedule of a multithreaded computation $\mathcal{X}$ is the maximum such value over the course of the execution, that is, $S_P(\mathcal{X}) = \max\{S(x) : x \in \mathcal{X}\}$. We define the *activation depth of a thread* to be the sum of the sizes of its own activation frame and the activation frames of all its ancestors. The *activation depth of the multithreaded computation*, referred to as S_1, then is the maximum activation depth of any of its threads. It is the minimum amount of space possible for a 1-processor execution of the computation in a depth-first manner. In particular we have $S_1(\mathcal{X}) \leq S_P(\mathcal{X})$.

For the time bound, we define the *work of the computation* T_1 as the number of tasks the computation is made up of. If we assume all tasks be unit-time tasks, then T_1 is the time it would take 1 processor to execute the computation, since 1 processor can execute only 1 task per time step. If we define the DAG depth of a task as the longest path in the DAG representation of the multithreaded computation that terminates at that task, the *DAG depth of the computation* is the maximum DAG depth of any task in the computation. It is denoted T_∞, since even with an infinite number of processors progress can always be made along the *critical path* of the computation—the path in the

DAG with its length equal to the DAG depth. Let $T_P(\mathcal{X})$ be the execution time of the computation with P processors using the execution schedule $\mathcal{X}$, then $T_P(\mathcal{X}) \geq T_\infty(\mathcal{X})$. Similarly, it should be clear that $T_1/P \leq T_P(\mathcal{X})$, since with P processors a maximum of P tasks can be executed at any time step. An upper bound on the execution time of a multithreaded computation using P processors can be obtained by *greedy schedules*.

Theorem 1: (Greedy-Scheduling Theorem) *For any multithreaded computation with work T_1, DAG depth T_∞, and for any number P of processors, any greedy execution schedule $\mathcal{X}$ achieves $T_P(\mathcal{X}) \leq T_1/P + T_\infty$.*

PROOF : We draw on the work of R. P. Brent [Bre74]. Suppose that s_i unit-time tasks are executed at step $i = 1, \ldots, T_\infty$ of a multithreaded computation using sufficiently many processors. Let $T_1 = \sum_{i=1}^{T_\infty} s_i$ be the total number of tasks to be executed. Using P processors, step i can be simulated in $\lceil s_i/P \rceil$ steps. Since $s_i = a_i P + b_i$ with $a_i = \lfloor s_i/P \rfloor$ and $b_i = (s_i - a_i P) \in \{0, 1, \ldots, P-1\}$, we obtain:

$$
\begin{aligned}
T_P &= \sum_{i=1}^{T_\infty} \left\lceil \frac{s_i}{P} \right\rceil = \sum_{i=1}^{T_\infty} \left\lceil \frac{a_i P + b_i}{P} \right\rceil = \sum_{i=1}^{T_\infty} \left\lceil \frac{a_i P + P + b_i - P}{P} \right\rceil \\
&= \sum_{i=1}^{T_\infty} \frac{a_i P + P}{P} + \sum_{i=1}^{T_\infty} \left\lceil \frac{b_i - P}{P} \right\rceil = \sum_{n=1}^{T_\infty} 1 + \frac{1}{P} \sum_{n=1}^{T_\infty} a_i P + \sum_{i=1}^{T_\infty} \left\lceil \frac{b_i - P}{P} \right\rceil \\
&= T_\infty + \frac{1}{P} \sum_{i=1}^{T_\infty} a_i P + \sum_{i=1}^{T_\infty} \left\lceil \frac{b_i - P}{P} \right\rceil \leq T_\infty + \frac{1}{P} \sum_{i=1}^{T_\infty} s_i + \sum_{i=1}^{T_\infty} \left\lceil \frac{b_i - P}{P} \right\rceil .
\end{aligned}
$$

From $b_i \in \{0, 1, \ldots, P-1\}$ it follows that $\lceil (b_i - P)/P \rceil \leq 0$. Using $\sum_{i=1}^{T_\infty} s_i = T_1$, we arrive at $T_P = \sum_{i=1}^{T_\infty} \lceil s_i/P \rceil \leq T_1/P + T_\infty$. □

From Theorem 1, the following two corollaries can be deduced:

Corollary 1: *The execution time $T_P(\mathcal{X})$ for any P-processor greedy execution schedule $\mathcal{X}$ of a multithreaded computation is optimal within a factor of 2.*

PROOF : According to Theorem 1, for any P-processor greedy execution schedule $\mathcal{X}$ of a multithreaded computation, the execution time $T_P(\mathcal{X})$ is bounded above by $T_P(\mathcal{X}) \leq T_1/P + T_\infty$. Then inequality $T_P(\mathcal{X}) \leq 2\max\{T_1/P, T_\infty\}$ also holds. $T_P(\mathcal{X}) \geq T_1/P$ and $T_P(\mathcal{X}) \geq T_\infty$ imply $T_P(\mathcal{X}) \geq \max\{T_1/P, T_\infty\}$, so that finally $\max\{T_1/P, T_\infty\} \leq T_P(\mathcal{X}) \leq 2\max\{T_1/P, T_\infty\}$. □

Corollary 2: *For any P-processor greedy execution schedule $\mathcal{X}$ of a multithreaded computation, linear parallel speedup is achieved when the number P of processors is no more than the average available parallelism T_1/T_∞.*

PROOF: According to Corollary 1, for any P-processor greedy execution schedule $\mathcal{X}$ of a multithreaded computation, the execution time $T_P(\mathcal{X})$ is optimal within a factor of 2. Let the number P of processors be no more than the average available parallelism T_1/T_∞, that is, $P \leq T_1/T_\infty$ or equivalently $T_\infty \leq T_1/P$. Now consider the proof of Corollary 1 and insert $T_\infty \leq T_1/P$ into $\max\{T_1/P, T_\infty\} \leq T_P(\mathcal{X}) \leq 2\max\{T_1/P, T_\infty\}$. Then $T_1/P \leq T_P(\mathcal{X}) \leq 2T_1/P$, that is, $T_P(\mathcal{X}) = \Theta(T_1/P)$, and linear parallel speedup is achieved. □

2.3.1. The Busy-Leaves Algorithm

In this subsection we consider the busy-leaves algorithm ('BL' for short) [BL99], the work stealing algorithm in Sec. 2.4 is based on—making its study be worthwhile in its own right.

Algorithm BL:

All living threads created before and during the execution of a strict multithreaded computation are maintained in a centralized thread pool that is uniformly available to all processors. Whenever a processor needs work it removes a ready thread, say, A from the pool, if there is any, and starts executing A's tasks until A either spawns, stalls, or dies.

1. If thread A spawns a child thread B, then A is returned to the thread pool, and its processor starts executing B.
2. If A stalls, then A is returned to the pool and its processor removes another ready thread from the pool and starts executing it.
3. If thread A dies, its processor checks whether A's parent thread, say, B has any living children. If not so, and if no other processor is executing B at that time step, then the processor takes B from the pool and starts executing it. Otherwise, the processor takes any ready thread from the pool.

If, for the sake of simplicity, we assume that processors do not contend with each other when accessing the pool, and if we further assume that threads can be added and removed from the pool in unit time, then the following theorem holds [BL99]:

Theorem 2: *For any strict multithreaded computation with work T_1, DAG depth T_∞, and activation depth S_1, and for any number P of processors, algorithm BL computes a P-processor execution schedule $\mathcal{X}$ with its execution time $T_P(\mathcal{X})$ satisfying $T_P(\mathcal{X}) \leq T_1/P + T_\infty$, and with space requirement $S_P(\mathcal{X}) \leq S_1 P$.*

PROOF: The bound on the execution time of the schedule $\mathcal{X}$ directly follows from the fact that algorithm BL computes a greedy execution schedule. Thus, Theorem 1 can be applied, and $T_P(\mathcal{X}) \leq T_1/P + T_\infty$. The space bound is a consequence of the fact that for strict computations, at any time during the execution, every leaf in the activation subtree has a processor working on it—*busy-leaves property*. By implication, the activation subtree has at most P leaves at any time during the execution. For every leaf, the space used by it and its ancestors is at most S_1. For the space requirement $S_P(\mathcal{X})$ of the computation we then get $S_P(\mathcal{X}) \leq S_1 P$. □

Theorem 2 states that algorithm BL can compute a strict multithreaded computation with linear expansion of space and linear parallel speedup, provided that the average available parallelism T_1/T_∞ is bounded below by the number P of processors.

2.4. A Randomized Work-Stealing Algorithm for Fully-Strict Multithreaded Computations

In this section we describe a randomized work stealing algorithm for fully-strict multithreaded computations ('WS' for short) with space requirements linear in the number P of processors and execution time linear in $1/P$ [BL99].

In algorithm WS the centralized thread pool used in algorithm BL is replaced by *P ready thread dequeues distributed to the P processors* of a parallel computer. Each processor is assigned exactly one dequeue containing threads that are ready for execution. Each dequeue has two ends, referred to as *head* and *tail*. For each dequeue only the processor it is assigned to (the *owner*) is allowed to add threads on the tail-end of the dequeue. The owner also takes threads from the dequeue's tail-end, using the dequeue in a LIFO manner. Threads that are migrated to other processors are taken (or *stolen*) from the head-end of the dequeue, making the dequeue appear as a FIFO for all other processors (called *thieves* in this context).

Algorithm WS:

Each processor maintains a dequeue of threads it is the owner of. A processor obtains work by removing a thread from the tail-end of its dequeue. If the dequeue contains any thread, say, A, it starts executing A until A either re-enables a stalled thread, spawns, dies, or stalls. The processor then proceeds as follows:

1. If thread A re-enables a stalled thread, say, B (A's parent), the now-ready thread B is added to A's processor's (empty) dequeue—why the dequeue is

empty at this time step is explained below—and the processor commences executing A. If thread A simultaneously re-enables a stalled thread, say, B and dies, its processor continues executing B and then A dies.

2. If A spawns a child thread B, its processor returns A on the tail-end of its dequeue and starts executing B.
3. If A stalls or dies, its processor checks its ready dequeue. If the dequeue is not empty the processor takes the thread on the tail-end and starts executing it. If the dequeue is empty, the processor becomes a thief and steals the thread on the head-end of the dequeue of another processor (called *victim* in this context) chosen uniformly at random—if the victim's dequeue is empty, the thief repeatedly chooses a new victim and tries again until either it successfully acquired a thread, or the computation has finished.

REMARK: Subitem 1 states that for thread A re-enabling a stalled thread B, A's processor's dequeue is empty at the respective time step.

First, thread B is A's parent due to the structure of a fully-strict multithreaded computation, and due to the busy-leaves property (see algorithm BL), which according to subitem 2 also applies to algorithm WS. All dependency-edges from A thus go to its parent, which then might be re-enabled by A.

Second, since thread B is found be stalled, and A's processor uses its dequeue in a LIFO manner, B must have been stolen by another processor that has executed B until it stalled. Also due to the structure of a fully-strict multithreaded computation, re-enabling B after the stall can only be done by B's child thread, which here is A. According to subitem 3, B is placed in the dequeue of another processor, as it was stolen, so that A's processor's dequeue must be empty at the respective time step.

Theorem 3: *For any fully-strict multithreaded computation with activation depth S_1, and for any number P of processors, algorithm WS computes an execution schedule $\mathcal{X}$ which uses at most $S_P(\mathcal{X}) \leq S_1 P$ space.*

PROOF: For the space bound we note that algorithm WS maintains the busy-leaves property. Thus, $S_P(\mathcal{X}) \leq S_1 P$ for any P-processor execution schedule of a fully-strict multithreaded computation. □

Theorem 4: *For any fully-strict multithreaded computation with work T_1, DAG depth T_∞, and for any number P of processors, algorithm WS computes a P-processor execution schedule $\mathcal{X}$ with execution time $T_P(\mathcal{X}) = \mathrm{O}(T_1/P + T_\infty)$ including scheduling overhead.*

PROOF IDEA: Since algorithm WS computes a greedy schedule $\mathcal{X}$, the proof of Theorem 4 is somehow similar to the proof of Theorem 2 except for the overhead explicitly addressed by Theorem 4—due to contention and communication. The latter point makes the proof of Theorem 4 a little more complicated. We therefore just give a proof idea based on [BL99], with some simplifications.

For the time bound we use an accounting argument: At each step of algorithm WS (execution schedule $\mathcal{X}$), we collect P dollars, one from each processor. If the processor executes a task at the respective step, the dollar goes to a 'work bucket'. If the processor initiates a steal attempt, the dollar goes to a 'steal bucket', and if the processor waits for a queued steal, then the dollar goes into a 'wait bucket'—it is assumed that a third party serially queues the work stealing requests. The running time bound is derived by bounding the number of dollars in each bucket at the end of the execution, summing these values up, and then dividing by P. Since the multithreaded computation consists of T_1 tasks, the work bucket contains T_1 dollars at the end of the computation. An upper bound for the number of dollars in the steal bucket follows from there are $\mathrm{O}(PT_\infty)$ stealing attempts over the course of the execution—along the critical path, which has length T_∞, at each step $\mathrm{O}(P)$ stealing attempts may occur, so that the total number of stealing attempts is $\mathrm{O}(PT_\infty)$.[3] In [BL99] it is shown that the expected number of dollars in the wait bucket is at most the number of dollars in the steal bucket. Adding up the dollars in the three buckets and dividing by the number P of processors gives $T_P(\mathcal{X}) = (1/P)\,\mathrm{O}(T_1 + PT_\infty) = \mathrm{O}(T_1/P + T_\infty)$. △

Cilk

The work stealing algorithm as presented in this section is at the core of the Cilk programming language [BJK+95]—from 2006 onwards advanced by Intel. Additional keywords like `cilk_spawn` and `cilk_sync`, to name the maybe most important ones, extend the C/C++ programming language and allow the programmer to express potential parallelism in its application. The Cilk runtime system then decides about how the respective portions of the code (threads) are mapped onto the underlying computer hardware, using the here considered work stealing scheme for work distribution.

Just for illustration purposes, Listing 2.1 illustrates the usage of Cilk for the recursive computation of the n-th Fibonacci number (pseudo-code). If for the thread `fib(n)` the predicate `n<2` evaluates to `false`, a new child thread `fib(n-1)` is spawned, and the respective processor commences with the execution of thread `fib(n-1)`, possibly

[3] In [BL99], the authors replace the DAG D describing the multithreaded computation by an augmented DAG D' which has DAG depth $T'_\infty \leq 2T_\infty$, where T_∞ is the DAG depth of D. The proof of Theorem 4 then is based on D'. However, our simplified argumentation stays the same.

```
function int fib(int n)
  if n<0 then return (−1)^(|n|+1) fib(|n|)
  if n==0 then return 0;
  if n==1 then return 1;
  int x=cilk_spawn fib(n−1);
  int y=fib(n−2);
  cilk_sync;
  return x+y;
```

Listing 2.1: Computation of the n-th Fibonacci number using Cilk (pseudo-code)

concurrently to the execution of thread `fib(n-2)` by another processor. At `cilk_sync` the two threads converge back to the execution of thread `fib(n)`.

In Chapter 3 we mimic this model for the execution of multithreaded computations (with dependencies) on the GPU and on Intel's Xeon Phi.

3. Work Stealing on GPU and Intel Xeon Phi

In this chapter we are concerned with the implementation and the evaluation of the work stealing scheme described in Chapter 2 on a current Nvidia GPU, and on Intel's Xeon Phi—for comparison with a standard x86 multi-core CPU, we also implement the scheme on the CPU. Beforehand, we need to introduce the hardware used hereafter, as its functioning imposes some constraints and limits on the implementation of the work stealing scheme as well as on the applications which later should use the scheme.

3.1. Nvidia Fermi GPU Architecture

The Fermi architecture is Nvidia's second unified shader[1] GPU architecture [Nvi12, Nvi09]. Fermi-based GPUs consist of up 16 processors (referred to as SMs), each of which equipped with 32 unified shader processors (referred to as SPs) which execute GPU programs, or portions of a GPU program, in SIMD (Same-Instruction Multiple-Data) manner. Hence, there is a total of up to 512 physical execution units. On the level of its SMs, the GPU can be thought of as MIMD machine with 16 cores and a per-core vector unit of width 32, similar to multi-core CPUs with SSE/AVX. Each SM is further equipped with 4 special function units for transcendentals, up to 48kB on-chip low latency shared memory, 16 load/store units for memory transfers between the GPU and its main memory, and 2 scheduling/dispatch units. SMs are organized into compute clusters containing 2 SMs each. On top of these compute clusters is a thread scheduling unit that is responsible for the creation of threads and the distribution of these threads to the compute clusters and SMs, respectively. Also there is a unified 768kB L2 cache that is shared by all SMs, and which is between the SMs L1 cache and the up to 6GB main memory. The entire memory subsystem has support for Error-Correcting Code—ECC. Figure 3.1 schematically illustrates such a GPU device.

From the programmers point of view the GPU appears as a *co-processor* that can

[1] A shader is an execution unit that executes shader programs written in an appropriate shader language. Today's GPUs are equipped with so-called unified shaders, which, by means of a unified instruction set, accumulate the functioning of the vertex-, pixel-, and geometry-shader along the graphs rendering pipeline in legacy GPUs.

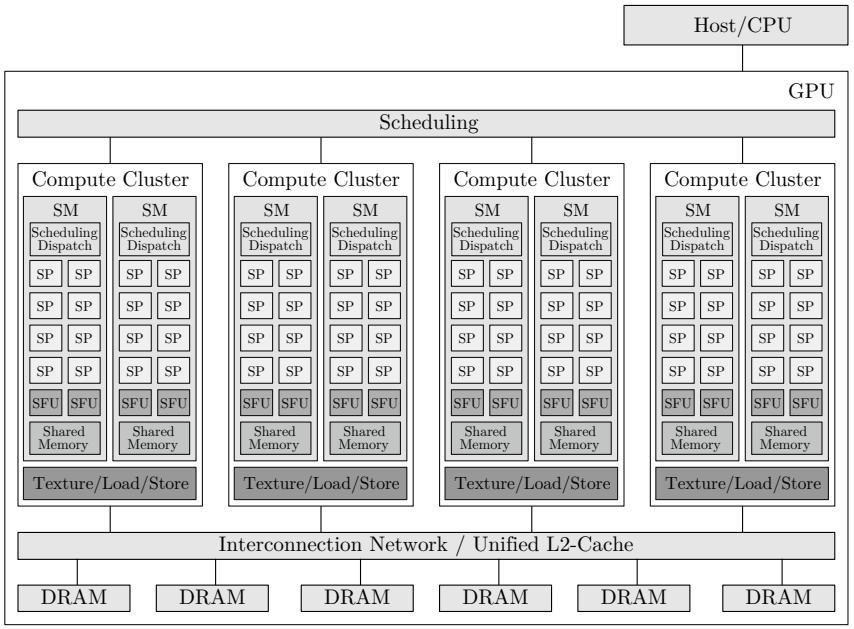

Figure 3.1.: Illustration of an Nvidia GPU based on the Fermi unified shader architecture (schematically). The image is based on illustrations found in [Nvi12, Nvi09].

be assigned work to from within adapted CPU programs (running on the host system the GPU is connected to via PCI-Express) using, for instance, Nvidia's CUDA API.[2] The programmer is responsible for explicitly initiating data transfers between the host system and the GPU—the host and the GPU have different physical address spaces—, and the invocation of GPU programs, called *kernels*. Kernels execute asynchronously to the host program. For an introduction to GPU computing using CUDA, the interested reader is referred to [Nvi12].

For this thesis, the most interesting point is how the GPU schedulers create and distribute threads to SMs and SPs, respectively. At the core of thread scheduling on Nvidia GPUs is a hierarchy of threads, organized into an up to three-dimensional grid of thread blocks, and up to three-dimensional thread blocks the grid is made up of. Thread blocks contain up to 1024 threads, organized into groups of 32 threads each, called *warps*. On Fermi GPUs each SM is capable of scheduling up to 1536 threads, that is, 48 warps, where the actual number of warps that concurrently reside on an SM depends on the resource consumption of each thread within a warp.[3] These up to 48 warps are distributed over up to 8 thread blocks.

[2] CUDA (Compute Unified Device Architecture) is an Nvidia proprietary programming model for Nvidia GPUs based on the unified shader architecture.

[3] There are 32768 registers per SM, and up to 48kB shared memory. For a total of 1536 threads to reside on an SM at the same time, each thread is restricted to use no more than 21 registers and 32 bytes of shared memory. If threads use more resources, the number of concurrent threads per SM goes down.

For each kernel launch the programmer defines the number of threads that should execute the kernel using an appropriate grid-block geometry. The GPU thread schedulers then successively create threads in units of thread blocks, and distribute these blocks to the GPU's SMs—if assigned to an SM, a thread block is not migrated to another SM. On the SMs the schedulers partition the blocks into warps and execute these warps on the SM's 32 SPs in SIMD manner (*wavefront execution*). The execution of the up to 48 warps per SM is done by means of *interleaved multithreading*. Switching between warps that are ready for execution is done by almost zero overhead according to some scheduling policy, which unfortunately is not made public by Nvidia. The idea is to hide (memory access) latencies—that may occur during thread program executions—by switching between warps that are ready for execution. If all warps within a thread block have finished their execution, the block as a whole is finished. For each finished thread block, the vacated SM is assigned a new thread block, if there is any. It should be clear that, due to the fact that usually it is not possible to schedule all threads defined by the programmer at the same time, threads each need to execute independently of each other, except for intra-thread-block communication and cooperation. Otherwise the behavior of a GPU program may depend on the execution order of the thread blocks.

3.2. Intel Many Integrated Core (MIC) Architecture

With the Larrabee project, Intel started the development of a many-core computer architecture [HKB12] that with the Xeon Phi [Int12] was officially announced very recently. While the original Larrabee architecture was designed for both graphics processing and HPC (High-Performance Computing), the current Xeon Phi MIC architecture is placed in the HPC sector only.

The Xeon Phi consists of an array of more than 50 Pentium-P5-based cores [Chr12], each of which augmented with 64-bit support, on-chip low latency L1 and coherent L2 cache, 32 512-bit vector registers, and 4-way interleaved multithreading. Using its vector unit, each core is capable of processing 16 32-bit words or 8 64-bit words per clock cycle. The cores, 16 memory controllers, and the PCI-Express client logic are connected by a bidirectional ring bus. Each direction of the ring consist of 5 independent rings: the data-block ring (64 bytes wide) for data transfers, 2 address rings which are used to send read/write commands and memory addresses, and 2 acknowledgment rings used to send flow control and coherence messages. Memory accesses first go through a tag directory (TD) which checks the L2 caches of all cores for the word requested, and forwards the request to the memory controllers only if the word is not present in any L2 cache.

Similar to the GPU, the Xeon Phi is connected to the host system via PCI-Express. Communication with the outside world is done using TCP/IP. The Xeon Phi runs a full

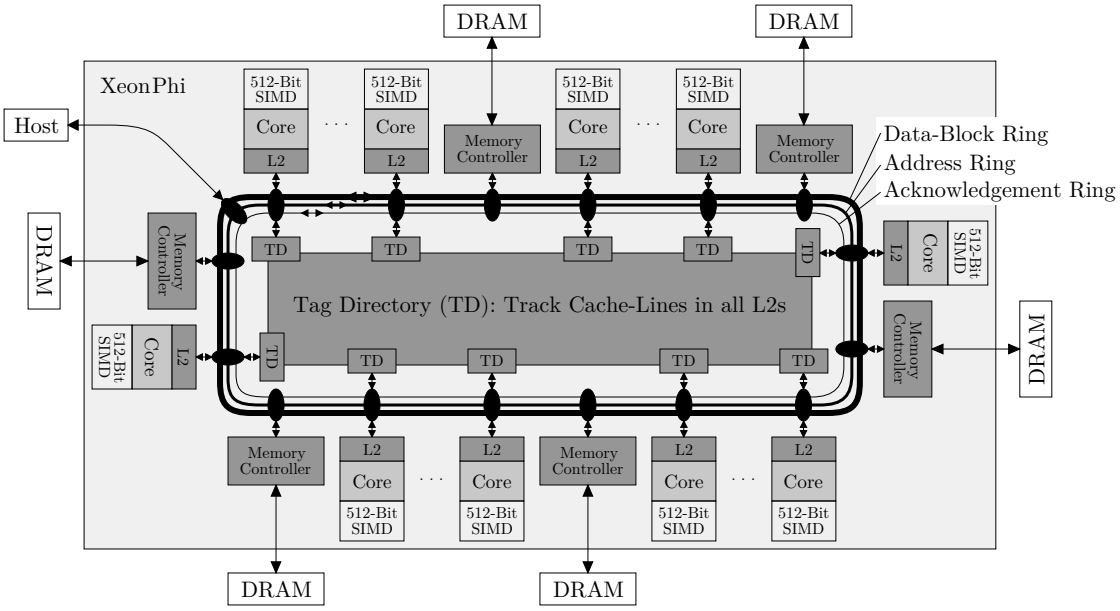

Figure 3.2.: Illustration of the Xeon Phi based on the Intel many integrated core (MIC) architecture (schematically). The image is based on illustrations found in [Chr12].

service Linux operating system which allows the programmer to use it as a (super-) computer in a computer. Within an application it can be used the same way as the GPU, but the Xeon Phi can also execute parallel applications natively without the need of a host program for kernel invocation. As it uses an x86 instruction set, the Xeon Phi is capable of executing (parallel) programs written for x86 CPUs, using OpenMP, pThreads, and Cilk, for instance. When writing programs/kernels for the 'Phi, almost all developer tools and programming languages known from x86 programming can be used. Applications that execute on the Xeon Phi are scheduled by its Linux operating system.

Table. 3.1 summarizes some properties of the hardware that is used in this thesis. All Information are either from the vendors' product database or from [Nvi09, Int12].

	Tesla M2090	Xeon E5-2670	Xeon Phi (Test-sample)
Processor Count P	16 (up to 768 logical)	8 (16 logical)	61 (244 logical)
SIMD Width	32	8	16
Clock Frequency	1.3GHz	2.6GHz	1.1GHz
Memory Size	6GB	-	8GB
Memory Bandwidth	177GB/s	51.2GB/s	352GB/s
Power Consumption	<225 W	<115W	<300W

Table 3.1.: Properties of Nvidia Tesla M2090, Intel Xeon E5-2670 and Intel Xeon Phi.

3.3. Problem Description

For both the GPU and the Xeon Phi the common approach for executing parallel applications is to partition the problem to be solved into subproblems that for the most part can be executed independently of each other. These subproblems then are expressed as thread programs which are mapped onto the processors according to some scheme. On the GPU dedicated scheduling units take on the mapping, whereas on the Xeon Phi (and the CPU) it is usually left to the programmer—iterating over the threads in chunks of the number of processors, for instance.

For multithreaded computations with dynamic thread creation, only few information about the work distribution at runtime are known beforehand. An obvious approach is to statically assign the portion of the threads that are created before the computation to the processors, and then at runtime to (re)distribute (newly created) threads to the processors in order to achieve an even load.

The GPU's scheduling units already implement some kind of load balancing on the level of the thread blocks. Since thread blocks are dynamically created at runtime, and then are assigned to available processors, even for applications with irregular workloads the scheduling should be balanced if the problem size in terms of thread blocks exceeds the processor count. As for a GPU program it is assumed that all work is known beforehand, dynamic thread creation cannot be handled by the GPU scheduler(s). Further it is assumed that on average threads behave identical so that all thread blocks created during the execution occupy the GPU's SMs for almost the same time. For SIMD applications this assumption applies. If threads dynamically create new threads, individual threads may require more time for their execution than others. The time the respective thread block resides on the SM then is possibly dominated by only a few warps in it. As the maximum number of thread blocks per SM is restricted to 8, a performance decrease might be observed when the thread scheduler(s) run out of warps they can switch between in the context of interleaved multithreading.

A viable solution to that issue is making thread blocks as small as possible, maybe containing just 1 or 2 warps. Unfortunately, the maximum number of concurrent threads per SM then is significantly below the maximum of 1536. Another approach is *dynamic load balancing*. For that to work a certain number of *persistent threads* is created so that hardware limitations are met (no more than $16 \times 1536 = 24576$ persistent threads on Fermi GPUs). Persistent threads that belong to the same thread group are represented by a master thread. Each group maintains its own work pool that is filled up with the statically assigned threads at the beginning of the computation. Persistent threads each execute a *super thread* that keeps them alive until the computation is done. The master thread executes the work stealing scheme in order to acquire work for its thread group.

With respect to the GPUs built-in load balancing strategy, performance improvements are to be expected, but the amount of the performance gain will strongly depend on the particular application. The overall performance of the application will also depend on the overhead introduced by the work stealing scheme.

On the Xeon Phi the static thread-processor assignment at the beginning of the computation may also result in some processors finish their threads significantly before other processors. The approach to handle these irregularities by means of dynamic load balancing is almost the same as for the GPU. The performance gain in so doing should be significantly above the one obtained on the GPU. The key difference to the GPU is that on the Xeon Phi there is no implicit load balancing by the hardware itself.

3.4. Multithreaded Data Structures

In this section we describe the data structures used for work stealing—blocking and non-blocking dequeue, in particular. We also evaluate the performance of the data structures and estimate the overhead for accessing the data.

Lock: Our implementation of a lock uses the atomic compare-and-swap (CAS) hardware primitive (all operations between << and >> are atomic, meaning they are executed without interruption):

```
function bool atomicCAS(address,expectedValue,newValue)
  bool success=false
  << if *address==expectedValue then *address=newValue,success=true >>
  return success
```

Listing 3.1: Atomic compare-and-swap (pseudo-code). `*address` is the value pointed to by `address`.

The `atomicCAS()` function compares the value pointed to by `address` with the value of `expectedValue`. If both are equal, the value pointed to by `address` is replaced by the value of `newValue` and the function returns `true` (success). Otherwise, it returns `false` (no success). On both Nvidia Fermi GPUs and x86-based processors an atomic compare-and-swap primitive is available. The lock itself is implemented as follows:

```
function void lock(address)
  while atomicCAS(address,0,1)==false do
    // spin around

function bool tryLock(address)
  return atomicCAS(address,0,1)

function void unlock(address)
  atomicEXCH(address,0) // Perform *address=0 without interruption
```

Listing 3.2: Lock based on atomic compare-and-swap (pseudo-code).

For the lock to work properly, the value pointed to by `address` (the value of the lock-variable) needs to be initialized to 0. The `lock()` function repeatedly compares the value of the lock-variable against 0, using the `atomicCAS()` primitive, until the lock has been acquired successfully—the value of the lock-variable then is 1. The `tryLock()` function tries to acquire the lock just once. The `unlock()` function sets the value of the lock-variable to 0—the lock is released.

3.4.1. Blocking Dequeue

The blocking dequeue (double-ended queue) enforces mutual exclusive access to its data elements by means of a lock. It provides the following operations: `push()`, `pop()`, and `steal()`. We assume the dequeue be array-based with array size N. Then the status of the dequeue can be derived from its head-end index and its tail-end index (see Fig. 3.3). The head-end index points to the next element in the dequeue that can be stolen, whereas the tail-end index points to the next empty slot a new element can be pushed to. The dequeue is empty if its head-end index and its tail-end index are equal. We define the dequeue be full, if its tail-end index is equal to the array size N, and if its head-end index is lower than its tail-end index—for the sake of simplicity, we do not allow the head- and tail-end index to swap around, that is, the dequeue is not cyclic.

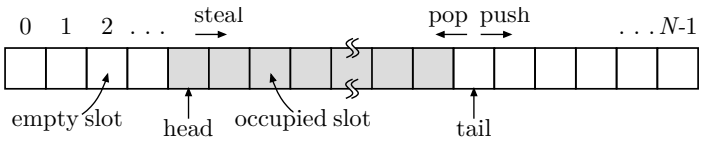

Figure 3.3.: Illustration of a dequeue for work stealing.

The `steal()` operation moves the head-end index one step towards the tail-end index, whereas the `pop()` operation moves the tail-end index one step towards the head-end index. Both of the two operations return one data element unless the dequeue is empty. If the dequeue is not full, the `push()` operation places a new data element into the dequeue at the position given by the tail-end index, and the tail-end index is moved one step towards the end of the array.

An implementation (in pseudo-code) of the blocking dequeue can be found in Listing 3.3. Appendix A.1 gives a concrete implementation for Nvidia GPUs. All functions start with acquiring a lock using either `lock()` or `tryLock()`. If the lock has been acquired, all operations until calling `unlock()` are done mutual exclusive. Thus, the dequeue is transferred from a consistent state to another one. We also list the function `stealChunk()`, which differs from the `steal()` function in that multiple elements can be stolen. By using `stealChunk()` instead of `steal()`, the effect of contention can be (partially) compensated.

```
struct dequeue<T> // T is a dummy for an actual data type
  T dq[0..N-1]
  int head,tail,lockVar=0 // The dequeue initially is unlocked

function bool push(T t)
  bool success=false
  lock(&lockVar) // &lockVar is the address of lockVar in main memory
  if tail<N then
    dq[tail]=t
    tail=tail+1
    success=true
  unlock(&lockVar)
  return success

function T pop()
  T t=NULL
  lock(&lockVar)
  if head<tail then
    tail=tail-1
    t=dq[tail]
    if tail==head then // Reset the dequeue. There is no ABA problem (see Sec. 3.4.2)
      head=0
      tail=0
  unlock(&lockVar)
  return t

function T steal()
  T t=NULL
  if tryLock(&lockVar)==true then
    if head<tail then
      t=dq[head]
      head=head+1
    unlock(&lockVar)
  return t

function T[] stealChunk(int n)
  T[] t=NULL
  if tryLock(&lockVar)==true then
    if head<tail then
      n=n>(tail-head)?(tail-head):n
      for i=0 to n-1 do
        t[i]=dq[head+i]
      head=head+n;
    unlock(&lockVar)
  return t
```

Listing 3.3: Blocking dequeue (pseudo-code). The notation `c=predicate?a:b` is a short form of **if** `predicate==true` **then** `a` **else** `b`.

The correctness of the implementation follows from only one processor can hold the lock at any time, and only this processor possibly changes the state of the dequeue. The implementation also guarantees deadlock-freedom as the lock is released eventually, with no side-effects affecting it.

3.4.2. Non-Blocking Dequeue

The non-blocking dequeue differs from the blocking one in that mutual exclusion is realized by means of atomic hardware primitives instead of a lock. The implementation given in Listing 3.4 is based on [ABP98].

```
union dqIndex
  int16 i16[2]     // Both i16[] and i32 refer to the same location in memory:
  int32 i32        // i16[0] is the head-end index, and i16[1] addresses the ABA problem

struct dequeue<T> // T is a dummy for an actual data type
  T dq[0..N-1]
  dqIndex head=0
  int tail=0

function bool push(T t)
  if tail<N then
    dq[tail]=t
    tail=tail+1
    return true
  return false

function T pop()
  T t
  dqIndex oldHead,newHead
  int oldTail
  if tail==0 then
    return NULL
①tail=tail-1
  t=dq[tail]
  oldHead=head
  if tail>oldHead.i16[0] then
    return t
  oldTail=tail
  tail=0
  newHead.i16[0]=0                  // Reset the dequeue
②newHead.i16[1]=oldHead.i16[1]+1   // ABA problem!
  if oldTail==oldHead.i16[0] then
③   if atomicCAS(&head.i32,oldHead.i32,newHead.i32)==true then
      return t
④head=newHead
  return NULL

function T steal()
  T t
  dqIndex oldHead,newHead
⑤oldHead=head
⑥if tail==oldHead.i16[0] then
    return NULL
  t=dq[oldHead.i16[0]]
  newHead.i16[0]=oldHead.i16[0]+1
  newHead.i16[1]=oldHead.i16[1]
⑦if atomicCAS(&head.i32,oldHead.i32,newHead.i32)==true then
    return t
  return NULL
```

Listing 3.4: Non-blocking dequeue (pseudo-code). The dequeue index `head` is of type `dqIndex` which is defined as a **union**. The reason for this is twofold: 1.) We need to address the ABA problem (see below), and 2.) on Nvidia GPUs the `atomicCAS()` primitive works on 32-bit and 64-bit words only.

A concrete implementation for Nvidia GPUs is given in Appendix A.2. We extended the non-blocking dequeue implementation in Listing 3.4 so that it also allows to acquire sets of elements of the warp size (32 on the Tesla M2090). Further, the implementation is not restricted to contain at most 65535 elements, but it allows for $2^{24} - 1$ elements.

Correctness

First, the value of the tail-end index `tail` is altered by the owner of the dequeue only. Since the owner either executes the `push()` function or the `pop()` function at any time, the `push()` function is not critical. It might occur that the value of `tail` is incremented by the owner concurrently to the execution of the `steal()` function by any of the thieves, but the worst thing that may happen is that the thieves evaluate the predicate `tail==oldHead.i16[0]` to `true` and return from `steal()`, even though the predicate becomes true at this point in time.

Second, as long as there is more than one element in the dequeue, the owner of the dequeue may execute the `pop()` function without using atomic primitives for the update of the dequeue state. The reason for that is the functioning of the atomic compare-and-swap primitive. Suppose there are n thieves that concurrently try to steal the head-end element of the dequeue. Each of these thieves takes a 'snapshot' of the dequeue's head-end index at the beginning of the `steal()` function (marker ⑤ in Listing 3.4). The thieves then remove the head-end element, determine the new state, and try to update the dequeue using the atomic compare-and-swap primitive (marker ⑦). The point is that for only one thief the head-end index of the dequeue has not changed in the meantime, and only for this thief the update is successful. After the atomic update of the dequeue, all other $n - 1$ thieves will find the dequeue with the updated head-end index, which then does not match the snapshot taken at the beginning of the `steal()` function. As a consequence, if n thieves concurrently try to steal the head-end element of the dequeue, the head-end index of the dequeue is moved just one step towards the tail-end index of the dequeue. Thus, if there is more than one element in the dequeue, and n thieves and the owner concurrently execute `steal()` and `pop()`, respectively, the owner does not contend with the thieves for the tail-end element of the dequeue, and therefore the owner can update the dequeue state without using atomic primitives.

Third, if the dequeue contains only one element then the owner of the dequeue and the thieves contend for this element. Before removing the element from the dequeue, the owner decrements the value of `tail` (marker ①). Suppose there are $n' \leq n$ thieves who evaluate the predicate ⑥ to `true`. These thieves return from `steal()` without success. The remaining $n - n'$ thieves compete with the owner, and due to the atomic update of the dequeue state (marker ③ and ⑦) only one of them achieves success. Since the dequeue then is empty, it is reset by the owner, either at ③ or ④.

The ABA Problem (Non-Blocking Dequeue only)

Suppose that any of the thieves executing the `steal()` function takes its snapshot of the dequeue's head-end index and then is preempted. If this thief comes back for execution the dequeue's head-end index may has changed, but there is the possibility that the thief's snapshot still matches with the current the head-end index. If the respective thief then has success with updating the dequeue's head-end index, it returns an element which previously was already returned by either the owner or another thief (in the meantime the dequeue might be reset multiple times). This so-called ABA problem is addressed by the introduction of a counter which is incremented each time the dequeue is reset by the owner (marker ②)—if `head` is a 32-bit word, the counter is represented by the upper 16 bits (see the **`union`** definition of `dqIndex` in Listing 3.4).

3.4.3. Performance Evaluation

In this subsection we evaluate the performance of the dequeue implementations on the GPU, the Xeon Phi, and a standard x86 multi-core CPU—see Tab. 3.1 for details on the hardware used. On the one hand we obtain information about the time it costs to access the dequeue's data elements, and on the other hand we can figure out what is the right choice for the respective compute device: blocking or non-blocking dequeue.

The way of proceeding is as follows: We use the GPU, the Xeon Phi, and the CPU in MIMD manner. For the GPU this means that thread groups (warps) each are represented and led by a master thread, and we just consider this master thread here (subsequently thread, for short). The benchmarking program creates n persistent threads that are executed by n processors. The number n will be varied over a meaningful range that is compatible with the number P of processors provided by the hardware. Each thread maintains a dequeue. Threads execute super threads in which they repeatedly acquire elements i) from their own dequeue only, and ii) from other thread's dequeues after they have finished their own dequeue. In setup iii) master threads also add new elements to their dequeue. That is, in setup i) they repeatedly execute the `pop()` operation, in setup ii) they execute the `pop()` and `steal()` operation, and in setup iii) all operations of the dequeue (`pop()`, `steal()`, `push()`) are executed. At the beginning of each iteration within the benchmark program, all dequeues are filled up with 32000 elements. In setup iii) every thread adds additional 32000 elements to its dequeue over the course of each iteration. When to execute `push()` and `pop()` is decided at runtime at random, with both operations be equiprobable.[4] An iteration completes when all elements from all dequeues have been acquired by the threads. The benchmarking program stops after a certain number of iterations.

[4]Each thread has its own linear congruential random number generator for that purpose.

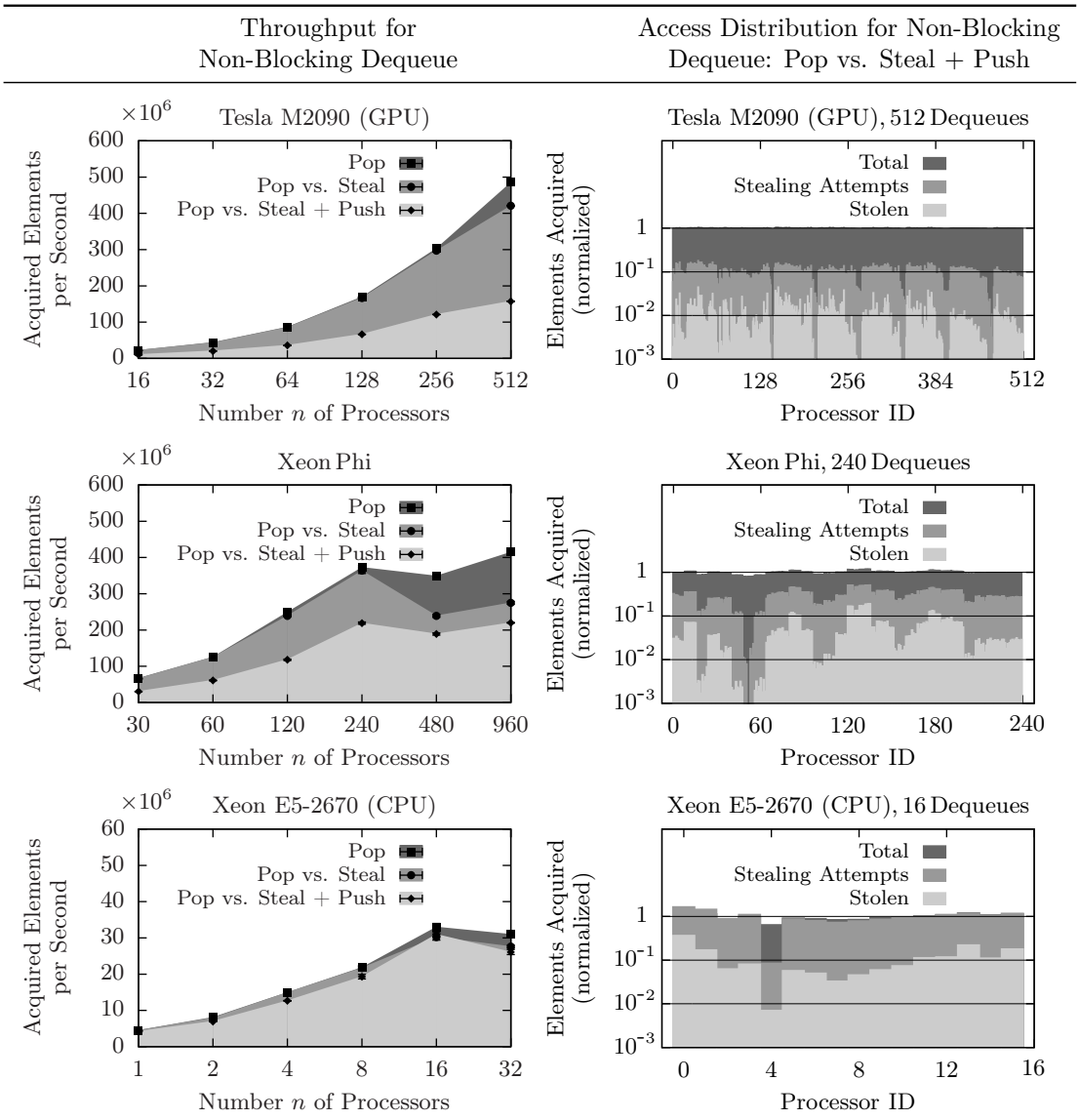

Figure 3.4.: Benchmarking results for the non-blocking dequeue implementation on GPU, Xeon Phi, and CPU. Values illustrated are averaged over 50 iterations within the benchmarking program. Values given in the right-hand side images are normalized to the expected number of elements per thread (50×64000).

Over the course of the benchmark we determine the overall throughput of the n dequeues, the number of acquired elements, the number of stealing attempts, and the number of actually stolen elements (the latter three, each per processor). Since the benchmarking program addresses the dequeue operations only, elements taken from any of the dequeues do not result in the execution of an associated thread. However, in a certain sense, the incrementation of the counters for the acquired/stolen elements and the stealing attempts can be understood as some kind of 'unit-time' tasks.

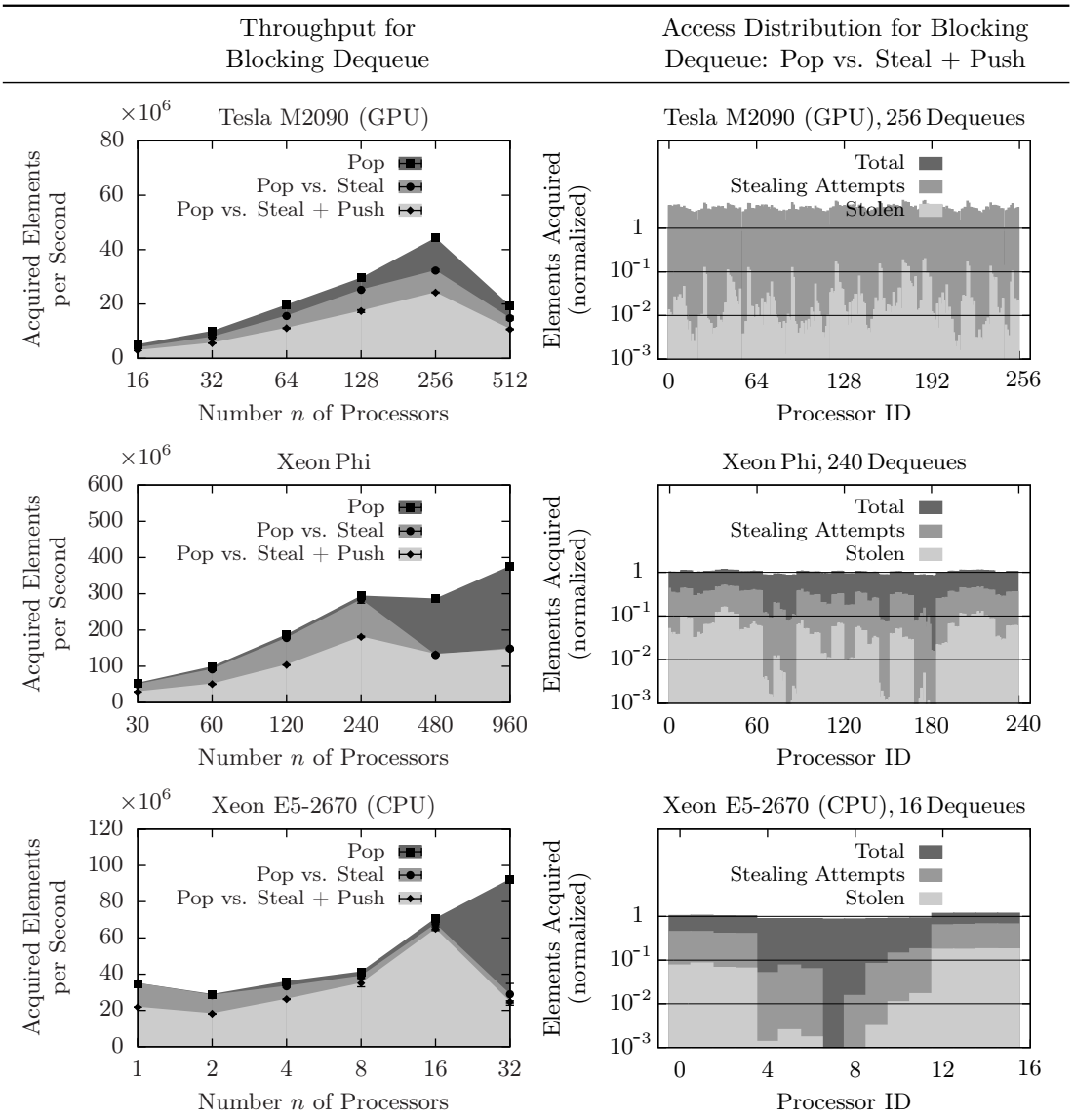

Figure 3.5.: Benchmarking results for the blocking dequeue implementation on GPU, Xeon Phi, and CPU. Values illustrated are averaged over 50 iterations within the benchmarking program. Values given in the right-hand side images are normalized to the expected number of elements per thread (50×64000).

We define the throughput of the n dequeues as the ratio of the overall acquired elements per iteration and the runtime of the respective iteration. The averaged throughput per dequeue then is the throughput divided by n. The inverse of this value gives an estimate of the averaged latency for accessing the dequeue's elements for each of the three setups. Figure 3.4 and 3.5 illustrate the benchmarking results averaged over 50 iterations for each setup and for each value of n. Error bars are included but almost always they are not visible—for almost all measurements taken the statistical error is below 1%.

The right-hand side images illustrate the number of elements acquired by each processor throughout 50 iterations of the benchmarking program.[5] The values are normalized to the expected number of elements per processor, which here is 50×64000. From these images it can be also seen that, due to the 'indeterminism' in the processors' push-and-pop behavior, some processors empty their dequeue before other processors—this models the execution of an actual computation with dynamic thread/task creation. These processors then become thieves and start stealing elements from the dequeues of other processors.

On the CPU, on average about 15% of the stealing attempts are successful, independently of whether the dequeue is blocking or non-blocking. On the Xeon Phi it is also about 15%. On the GPU only the non-blocking dequeue seems to work well. Here, on average about 10% of the stealing attempts are successful. Due to the stealing, thief threads acquire more elements than expected, whereas for threads from which elements are stolen the reverse is true. The normalized number of acquired elements thus varies around 1 for all threads.

An interesting point in this respect is that seemingly almost always the same threads are the ones elements are stolen from. On the GPU we assume that the internal scheduling of the threads (the warps) is not fair, so that some threads are preferred over others.

On the Xeon Phi this argument should not apply as here the hardware performs a strict per clock cycle round-robin switching between threads that run on the same physical execution unit. Rather, we assume that if there is heavy traffic on the ring bus, situations may occur where requests send over the ring bus stall. As memory is allocated at the beginning of the benchmark program, it might be possible that a stall affects always the same threads when trying to update their data structures concurrently to other threads. Maybe there are also NUMA (Non-Uniform Memory Access) effects. Also consider that the use of atomic primitives for synchronization methods results in memory bus locking and cache invalidation, both possibly affecting individual threads more than others.

The left-hand side images confirm non-blocking synchronization be more suitable for massively parallel computers than blocking synchronization. While on the GPU this statement is quiet obvious, on the Xeon Phi both the non-blocking and the blocking dequeue implementation seem to work well. On the CPU the blocking dequeue is even superior to the non-blocking dequeue. However, for all three compute devices, the overall throughput of the dequeues scales almost linearly with number n of processors as long as n is below or equal to the device's processor count P (see Tab. 3.1).

[5] During the execution of the benchmarks, we checked the correct functioning of the implementations of the dequeue: the number of elements put into the dequeues was exactly the number of elements that were concurrently removed from the dequeues by the processors.

When comparing the throughput for the three different setups, the highest throughput is obtained if there is no stealing and no pushing (setup i)). In setup ii), where a certain number of `pop()` operations is executed concurrently to `steal()` operations, the throughput decreases. In the case of the non-blocking dequeue, the reason for that is the use of atomic primitives in the `steal()` operation—atomic operations enforce memory bus locking while the operation is performed, and cache invalidation afterwards. In the case of the blocking dequeue, `steal()` operations make the owner of the respective dequeue contend with thief threads for the dequeue's lock. In both cases the throughput goes down. In setup iii) the decrease in the throughput is due to half of the `pop()` and `steal()` operations require the `push()` operation has been executed previously—elements taken from the dequeues must have been added to them previously. The effective costs of a `pop()` operation and a `steal()` operation, respectively, then increase by the costs of a `push()` operation. Again the throughput goes down.

Table 3.2 summarizes some characteristic (averaged) per-dequeue values extracted

Characteristics for Dequeue:			Non-Blocking	Blocking
Setup i)	Throughput in	GPU:	0.95	$0.17^{\dagger}$
	10^6 Elements/Second	Xeon Phi:	1.55	1.22
		CPU:	2.06	4.42
	Latency in	GPU:	1.05	$5.88^{\dagger}$
	10^{-6} Seconds/Element	Xeon Phi:	0.65	0.82
		CPU:	0.49	0.23
Setup ii)	Throughput in	GPU:	0.82	$0.13^{\dagger}$
	10^6 Elements/Second	Xeon Phi:	1.52	1.18
		CPU:	1.89	4.23
	Latency in	GPU:	1.22	$7.69^{\dagger}$
	10^{-6} Seconds/Element	Xeon Phi:	0.66	0.85
		CPU:	0.53	0.24
Setup iii)	Throughput in	GPU:	0.31	$0.09^{\dagger}$
	10^6 Elements/Second	Xeon Phi:	0.91	0.75
		CPU:	1.94	4.06
	Latency in	GPU:	3.22	$11.1^{\dagger}$
	10^{-6} Seconds/Element	Xeon Phi:	1.10	1.34
		CPU:	0.52	0.25

Table 3.2.: Benchmarking result for the blocking and the non-blocking dequeue. Values listed are for setup i), ii), and iii), where the number n equals the number of logical processors provided by the hardware (see Tab. 3.1). Note: Values are per dequeue. Example: The throughput for setup i) on the GPU using 512 dequeues is about 485×10^6 Elements/Second. The averaged per-dequeue throughput then is 0.95×10^6 Elements/Second. The Latency is the inverse of this value.
†We use 256 processors here.

from the benchmarking results. On the GPU the non-blocking dequeue should be used, whereas on the CPU the blocking dequeue is the matter of choice. On the Xeon Phi both of the two seem to be suitable. We also considered the implementation of the blocking dequeue with the `stealChunk()` operation. On the GPU and the Xeon Phi there was only a small gain compared to using the `steal()` operation. On the CPU a performance improvement over the blocking dequeue (using the `steal()` operation) of about a factor 1.4 can be achieved.

3.5. Scheduling Multithreaded Computations on GPU and Xeon Phi

In this section we implement the work stealing scheme as detailed in Sec. 2.4. We first describe our approach to dependency resolution, and then we evaluate the implementation using a synthetic application.

3.5.1. Dependency Resolution

A multithreaded computation consists of threads that are connected by spawn and data-dependency edges (see Sec. 2.1). For fully-strict multithreaded computations, dependency edges exist between parent and child threads only, going from the child to the child's parent. If both the child and the parent thread execute concurrently—the parent thread must haven been stolen for this situation to occur—two cases can be distinguished:

1. The parent does not depend on its child thread. It then can be executed independently of its child thread.

2. The parent depends on its child thread, that is, it contains a task with a data-dependency edge incident on it. The parent thread stalls at this task if the dependency is not resolved. Otherwise, the execution continues until either the parent thread dies or it stalls at any of the following tasks.

Since the first case is somehow trivial, the second one is challenging as the parent thread might stall. In this case it must be possible for the child thread to continue executing the parent thread. Especially on the GPU we run into difficulties with making threads stop at some point in their execution and to substitute these threads by others which then continue the execution.

In our approach to dependency resolution we split up threads into subthreads with dependency counters each. Figure 3.5 illustrates the splitting for a function f into f_1, f_2, and f_3, with a spawn and a synchronization point within. f then is the composition of f_1, f_2, and f_3, that is, $f = f_3 \circ f_2 \circ f_1$. The `spawn()` is placed at the end of f_1,

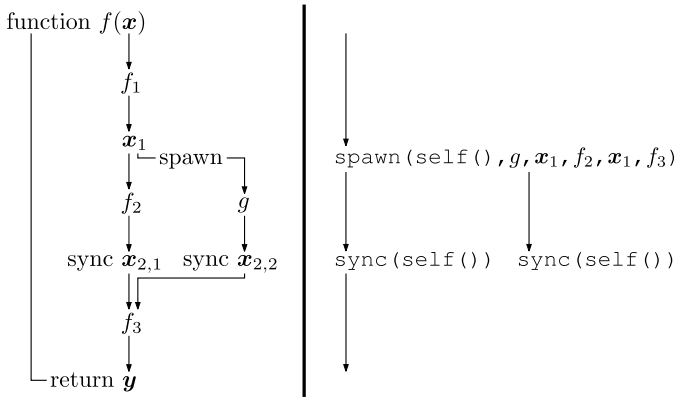

Figure 3.6.: Function f expressed as a composition of the functions f_1, f_2, and f_3 with a spawn between f_1 and f_2, and a synchronization point between f_2 and f_3. The right-hand side sub-image illustrates the usage of the `spawn()` and `sync()` functions given in Listing 3.5 in this context.

and the synchronization point is placed at the end of f_2 and g, respectively, where g is the previously spawned function. The functions f_1, f_2, f_3, and g correspond to the aforementioned subthreads (threads hereafter, for short).

According to Algorithm WS (see Sec. 2.4), the thread containing function g is executed by the thread that did the spawn, and the thread containing f_2 is added to its (processor's) dequeue. The point where both threads converge back to the execution of thread f_2 is implemented by the `sync()` function. Listing 3.5 describes the operation of `spawn()` and `sync()` in C pseudo-code.

The `spawn()` function is given a reference to the calling thread, function pointers to the next-to-the-current function and the spawned function (f_2 and g in the example above), as well as input arguments to these functions. Within the `spawn()` function threads t_1 and t_2 are created that encapsulate the functions given to `spawn()`. Thread t_1 is executed by the calling thread's processor itself in any case, and t_2 may be executed by any other processor. For this purpose t_2 is added to the calling thread's (processor's) dequeue. If the dequeue is full, even t_2 is executed by the calling thread's processor right after the execution of thread t_1 (depth-first execution).

The optional sixth (`dependent`) and seventh parameter (`abSyncFunc`) are to signalize a dependency between the two threads t_1 and t_2. If the sixth parameter is set to `true` (the default value is `false`), a third thread `syncThread` is created that acts as synchronization object—`syncThread` is stored in the heap memory so that it persists the execution of the `spawn()` function. Both t_1 and t_2 have a reference to `syncThread`, so that accessing the `syncThread` object can be done in O(1) time. `syncThread` itself has a dependency counter `depCount`, which is set to 2 (as there are 2 threads that synchronize to each other using `syncThread`), and a function pointer set to the `abSyncFunc` function given to `spawn()` (f_3 in the example above). The input

```
struct thread
  funcPtr func
  arguments args
  thread *syncThread
  int depCount
  int lockVar=0

function void spawn(thread *currentThread,
                    funcPtr aFunc,arguments aArgs,
                    funcPtr bFunc,arguments bArgs,
                    bool dependent=false,
                    funcPtr abSyncFunc=NULL)
  thread *syncThread=NULL
  if dependent==true then
    syncThread=new thread // create synchronization thread visible outside the function scope
    syncThread←(abSyncFunc,NULL,currentThread->syncThread,2)
  // create actual threads: t1 is the spawned thread
  thread t1←(aFunc,args=aArgs,syncThread,0) // t1 can be immediately executed
  thread t2←(bFunc,args=bArgs,syncThread,0) // t2 can be immediately executed
  // try to add thread t2 to the dequeue
  if myDequeue.push(t2)==true then
    execute t1->func
  else // dequeue is full! execute both threads by myself
    execute t1->func // Depth-first execution of t1
    execute t2->func // and t2

function void sync(thread *currentThread)
  if currentThread->syncThread!=NULL then
    lock(&currentThread->syncThread->lockVar) // acquire syncThread's lock
    currentThread->syncThread->args='some modifications'
    currentThread->syncThread->depCount-=1
    if currentThread->syncThread->depCount==0 then // all dependencies resolved
      unlock(&currentThread->syncThread->lockVar) // release lock
      execute currentThread->syncThread->func
      delete currentThread->syncThread // delete synchronization thread
    else
      unlock(&currentThread->syncThread->lockVar) // release lock
```

Listing 3.5: Implementation of the `spawn()` and `sync()` function (pseudo-code).

arguments to `syncThread`'s function are created by t_1 and t_2 when calling the `sync()` function. If t_1 and t_2 do not depend on each other, the `sync()` function has no effect. Otherwise, the calling threads acquire the `syncThread`'s lock-variable, and modify `syncThread`'s state. In particular, each thread decrements the value of `depCount` by one. The thread for which `depCount`'s value is zero after the decrementation resolves the dependency. Its processor then starts executing the thread/function pointed to by `syncThread`'s function pointer, and afterwards the `syncThread` object is deleted.

3.5.2. Evaluation of the Implementation

For the evaluation of the implementation of the work stealing scheme we use two synthetic application scenarios. The first scenario (SC 1) models the situation of a recursive function call. That is, a thread duplicates itself, using the `spawn()` function, until the recursion stops, say, after N recursive calls. On each level of the recursion a synthetic kernel is executed that keeps the executing processor busy for a certain amount of time.[6]

Figure 3.7 illustrates the procedure for a kernel that has return value 1—we implemented a kernel that performs k ADD and $k-1$ SUB operations, using assembly language to prevent the compiler from any optimization. Threads that synchronize add up these return values for a final value of $N + 1$.

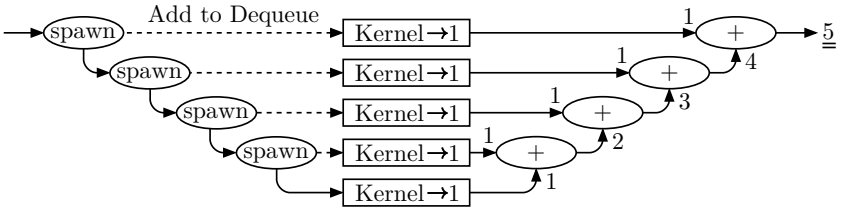

Figure 3.7.: Synthetic application for the evaluation of the implementation of the work stealing scheme with dependencies. Here the number of recursive calls is $N = 4$.

Over the course of the recursive subroutine calls, the initial thread (the one that performs the recursion) adds N threads into its (processor's) dequeue. The total of $N+1$ threads is executed by $1 \leq n \leq N + 1$ processors using the work stealing scheme. The value of N is chosen such that the number P of processors on the device used is at least $N + 1$. The device then can be thought of as being equipped with an infinite number of processors, and we should see the execution time of the application be bounded below by the execution time of the kernel T_{kernel}. This value approximates T_∞, where $T_{\text{kernel}} \gtrapprox T_\infty$—increasing the number of processors does not speed up the application. For values $n < N + 1$, the execution time T_n is $T_n \gtrapprox \lceil (N + 1)/n \rceil \, T_{\text{kernel}}$.

Figure 3.8 shows application runtimes with recursion depth $N = 29$ on the Xeon Phi and $N = 31$ on the Tesla M2090, respectively, and with $T_{\text{kernel}} = kT_{\text{ADD}} + (k - 1)T_{\text{SUB}}$ and $k \in \{100, 1000, 10000, 100000\}$. T_{ADD} and T_{SUB} are the times it costs to perform an ADD and a SUB operation, respectively. As T_{ADD} and T_{SUB} vary from device to device, we decided to normalize runtimes to the execution time of the kernel. All values given are averaged over 6 runs of the benchmarking program. It can be seen that our measurements

[6] Nvidia GPUs of device capability at least 2.0 have support for recursion on the level of `__device__` functions. In the CUDA programming model GPU functions are divided into `__global__` functions (kernels) that are callable from within the host program, and `__device__` functions which are per-thread functions that are callable from within `__global__` functions. The CUDA runtime system provides the `cudaDeviceSetLimit()` function which allows to set the per-thread stack size.

almost ideally match the expected values if T_{kernel} is sufficiently large ($k \gg 10000$). If T_{kernel} becomes too small, the overhead for the creation of the synchronization threads, and the synchronizations themselves overcompensate the performance increase obtained with the load balancing scheme.

For all benchmarks we checked the correctness of the output. We found no discrepancies.

In the second application scenario (SC 2) each processor initially is assigned one thread. During the execution of this thread new threads are created in the same way as in scenario SC 1 with the recursion depth N given by the processor ID, ranging from 0 to $n-1$. The total amount of work $W_1(n)$ (in terms of threads) then is $W_1(n) = \sum_{i=0}^{n-1} T_{1,i} = \sum_{i=0}^{n-1}(i+1) = n(n+1)/2$, where $T_{1,i}$ is the amount of work (in terms of threads) assigned to processor i. When using the work stealing scheme (WS), $W_1(n)$ should be distributed across the n processors such that the maximum number of threads executed by each processor is $\lceil n(n+1)/(2n) \rceil = \lceil (n+1)/2 \rceil$. The execution time T_n of the program then is $T_n = \lceil (n+1)/2 \rceil T_{\text{kernel}}$, and the speedup $\mathcal{S}_n$ over the execution not using work stealing (no WS) is $\mathcal{S}_n = \max\{T_{1,i} : i \in \{0,1,\ldots,n-1\}\} T_{\text{kernel}} / (\lceil (n+1)/2 \rceil T_{\text{kernel}}) = nT_{\text{kernel}} / (\lceil (n+1)/2 \rceil T_{\text{kernel}}) = n / \lceil (n+1)/2 \rceil$.

Figure 3.9 shows application runtimes on the GPU and the Xeon Phi, with $T_{\text{kernel}} = kT_{\text{ADD}} + (k-1)T_{\text{SUB}}$ and $k \in \{100, 1000, 10000, 100000\}$. Again, all values given are averaged over 6 runs of the benchmarking program. The number n of processors was varied over $\{1, 2, \ldots, 64\}$ on the GPU—we had 4 threads (warps) per thread block, so that up to 16 thread blocks are scheduled, matching the number of processors on the GPU—, and $\{1, 2, \ldots, 60\}$ on the Xeon Phi, where threads were pinned to processors. The expected execution times are met for sufficiently large k.

Comparison against Cilk

In order to compare our implementation of the work stealing scheme with Cilk [BJK+95], the `spawn()` and the `sync()` function are replaced by their Cilk pendants. For the runtime measurements to be reliable and reproducible, we execute the entire computation twice, for one thing before starting the timer, and for another thing after having started the timer—in this way we found the overhead due to thread creation become negligible, as all threads are already created during the first run and then can be immediately used in the second run. Similar to the benchmarking procedure of our implementation of the work stealing scheme, for setups SC 1 and SC 2 execution times are averaged over 6 runs of the benchmarking program.

Figure 3.10 illustrates the benchmarking results. The execution times for the 'no-work-stealing' runs of our implementation are taken as reference. Seemingly, something goes terribly wrong. For small kernels with $k < 100000$ there is almost no speedup over

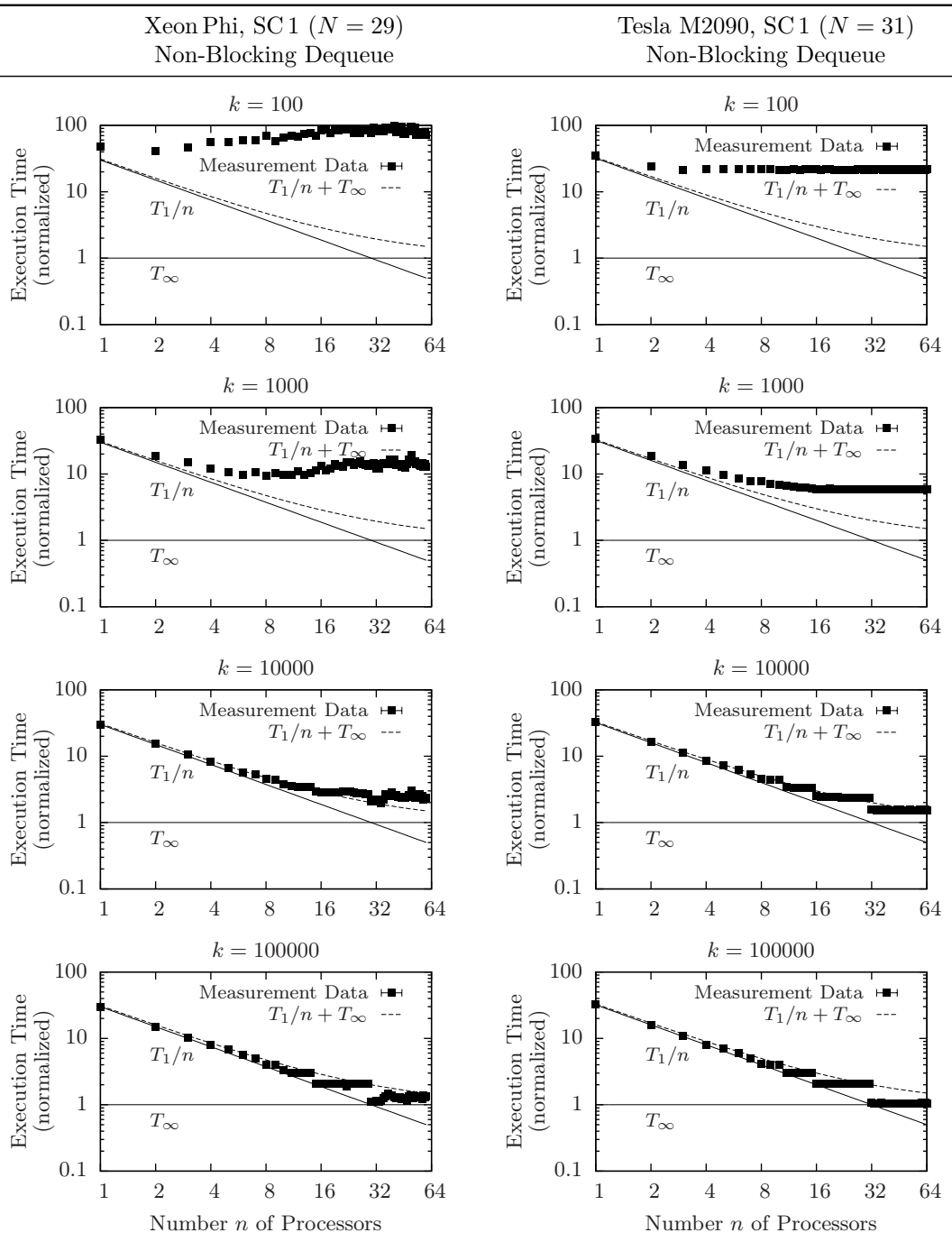

Figure 3.8.: Runtimes of an application (SC 1) modeling recursive subroutine calls with recursion depth $N = 29$ on the Xeon Phi, and $N = 31$ on the Tesla M2090, respectively. The processor that executes the recursive thread adds N threads to its dequeue during the execution. All $N + 1$ threads are executed by up to 60 (Xeon Phi) and 64 (Tesla M2090) processors, respectively. Runtimes are normalized to the execution time T_{kernel} of the kernel that is executed on each level of the recursion. Each kernel performs k ADD operations and $k-1$ SUB operations. The value of k was chosen to $k \in \{100, 1000, 10000, 100000\}$.

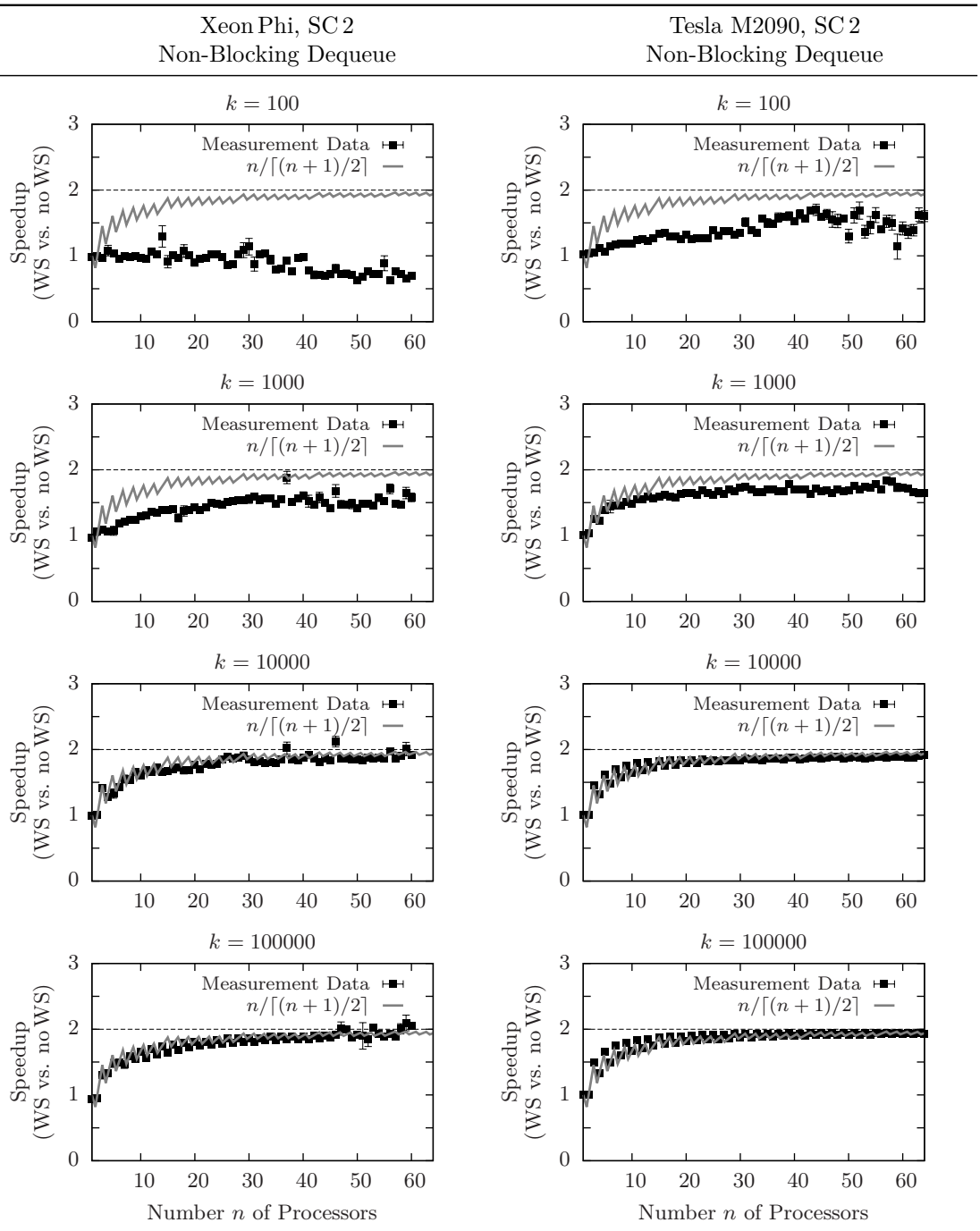

Figure 3.9.: Speedup 'work-stealing (WS) vs. no-work-stealing (no WS)' for application scenario SC 2. Each of the n processors executes a thread running the application described above (SC 1) with recursion depth $n-1$. The expected speedup when using the work stealing scheme is $n/\lceil(n+1)/2\rceil$—the gray solid line in the plots. On each level of the recursion a kernel is executed performing k ADD operations and $k-1$ SUB operations. The value of k was chosen to $k \in \{100, 1000, 10000, 100000\}$.

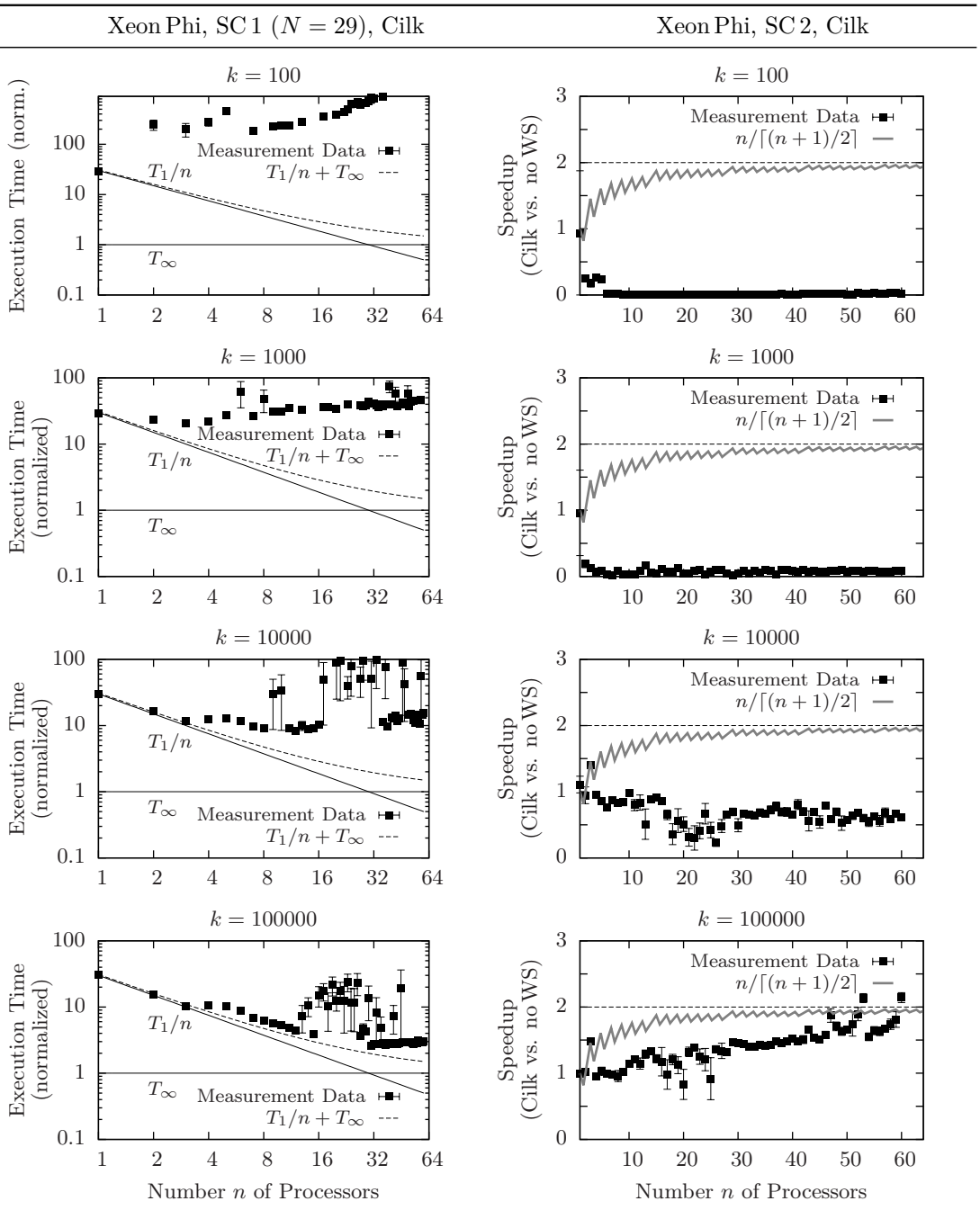

Figure 3.10.: Runtimes for application scenario SC 1 on the Xeon Phi using Cilk, and speedups 'Cilk vs. no-work-stealing (no WS)' for application scenario SC 2. For further details, see Fig. 3.8 and Fig. 3.9.

the 1-processor execution of SC 1, and also for SC 2 execution times are far away from what is expected. We told the Cilk runtime system to use no more than n workers —we called `__cilkrts_set_param("nworkers", NUM_PROCESSORS)` before any

computation. Only for $k > 100000$ the Cilk implementation seems to yield the expected runtimes.

Since we are not the experienced Cilk programmers, it might be possible that some modifications of the benchmarking program will increase its performance.

4. Application Scenarios

In this chapter we apply the work stealing scheme to real-world problems. By comparing execution times of programs using work stealing against those not using this scheme, we illustrate the necessity of dynamic load balancing on (massively) parallel computers when computations become irregular in workload.

In the first section we therefore take up the ray tracing method for image creation, and in the second section we focus on the breadth-first search algorithm. While for ray tracing there is already sufficient potential for parallelism at the beginning of the computation, breadth-first search starts up with work for one processor only and develops (massive) parallelism during the execution.

4.1. Ray Tracing

In this section the work stealing scheme is merged together with an already existing implementation of a ray tracer on the GPU that was developed within the scope of a software project course on parallel algorithms on GPUs at Freie Universität Berlin. At first, we give a brief overview on ray tracing and summarize our contribution to the course. Then we move on to the integration of load balancing into the ray tracer.

4.1.1. The Ray Tracing Method

The ray tracing method aims for the production of realistic high-quality images of a scene described by geometric primitives such as triangles, spheres, etc. The image is created pixel by pixel. For each pixel a ray of light, leading from the observer's eye to that pixel, is traced through the scene. The most portion of the work is in the detection of possible intersection points between the rays and the geometric primitives that are the closest ones with respect to the pixels the rays are associated with. For these intersection points, the evaluation of the Phong lighting equation (we do not introduce to computer graphics; see [SM03, SAG+05], for instance) allows for the determination of the color values of the respective pixels in the first order. Figure 4.1 illustrates the procedure.

In order to incorporate reflectivity and refraction, higher-order rays (secondary rays, for short) need to be considered. These rays have their origin in the aforementioned intersection points, and their direction result from the laws of geometric optics. Secondary

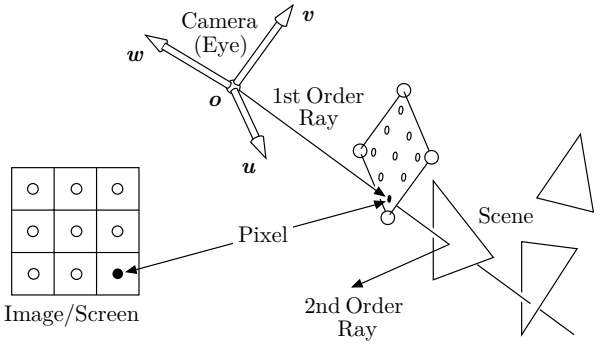

Figure 4.1.: Tracing a ray through a scene. The geometric primitive (here a triangle) which is the closest one determines the color of the respective pixel. The image originates from a drawing in [SAG+05].

rays are traced through the scene in the same way as first-order rays, yielding a hierarchy of rays for each pixel. On each level of this hierarchy the color value conveyed by the ray on that level can be determined by evaluating Phong's lighting equation. The color value of a pixel is computed as the sum of the color values of all its contributing rays.

Listing 4.1 illustrates the per-ray execution in pseudo-code, with some annotations with respect to Sec. 4.1.3 in lines 12 and 16. The scene is considered to be made up of triangles only. The interested reader is referred to [SM03, SAG+05, WF12].

```
function rgb rayColor(ray r,rgb weight,int recursionDepth)
  if weight< W_cutoff or recursionDepth==maxRecursionDepth then
    return black // RGB representation: (0,0,0)
  // Initialize color value
  rgb color=ambientColor
  // Determine intersection point with any of the scene's triangles
  triangle t=intersectionRoutine(r)
  if t!=NULL then // If there is an intersection point then...
    if t.reflectivity!=0 then // Reflexion
      rRefl ← 'create reflexion ray'
      color+=rayColor(rRefl,t.reflectivity,recursionDepth+1)
12:   // MTC: color+=spawn rayColor(rRefl,weight*t.reflectivity,...+1)
    if t.refractivity!=0 then // Refraction
      rRefr ← 'create refraction ray'
      color+=rayColor(rRefr,t.refractivity,recursionDepth+1)
16:   // MTC: color+=spawn rayColor(rRefr,weight*t.refractivity,...+1)
    for light ∈ {visible light sources} do
      color+='evaluate Phong's lighting equation (see [SAG+05], for instance)'
  return weight*color
```

Listing 4.1: Per-ray execution in the context of ray tracing (pseudo-code). `rgb`, `ray`, and `triangle` refer to complex data types representing RGB color values, rays, and triangles, respectively. Lines 12 and 16 contain annotations for the execution within a multithreaded computation (MTC). Since threads are independent of each other, there are no synchronization points.

4.1.2. Implementation Details

The implementation of the ray tracing method and its parallelization is straightforward for the most part as there are no dependencies between the rays (this allows a trivial parallelization). One of the challenging parts of the implementation is the representation of the scene that should be rendered. Usually, there are thousands of triangles, with every one of them being a potential candidate for having an intersection point in common with any of the rays. Testing the triangles for intersection in brute force manner, that is, for each ray all triangles are considered, then would take $T(n_r, n_t) = \Theta(n_r n_t)$ time for n_r rays and n_t triangles.

A better way is to partition the volume occupied by the scene into disjoint subvolumes containing a subset of the scene's triangles each, and to repeat this procedure within these subvolumes recursively until, for instance, the number of triangles in the subvolume is below a cutoff value, say, n^*. In this way, a hierarchy of subvolumes (bounding boxes) is created. The $\Theta(n_r n_t)$ brute force runtime reduces to $\mathrm{O}(n_r h(n_t))$ ray-bounding-box intersection tests, and possibly some ray-triangle intersection tests within subvolumes for which the ray hits the subvolume's bounding box. $h(n_t)$ is the height of the tree given by the bounding box hierarchy. The number of ray-triangle intersection tests within any of the bounding boxes is bounded above by n^*. Since n^* does not depend on n_r nor on n_t, the per-box ray-triangle intersection tests are in $\mathrm{O}(1)$. Hence, $T(n_r, n_t) = \mathrm{O}(n_r h(n_t))$.

For our implementation we use an octree data structure with axis-aligned bounding boxes for the partitioning of the scene [SM03].

GPU Implementation

On the GPU the octree data structure in its explicit recursive definition is unsuitable for the scene traversal. Although current Nvidia GPUs of the Fermi architecture (and later) have support for recursion, a transformation of the data structure into a non-recursive one seems to be meaningful. We therefore replace the octree data structure by a pre-ordered linked list [WF12]. Figure 4.2 illustrates our approach for a binary tree (the procedure is similar for the octree).

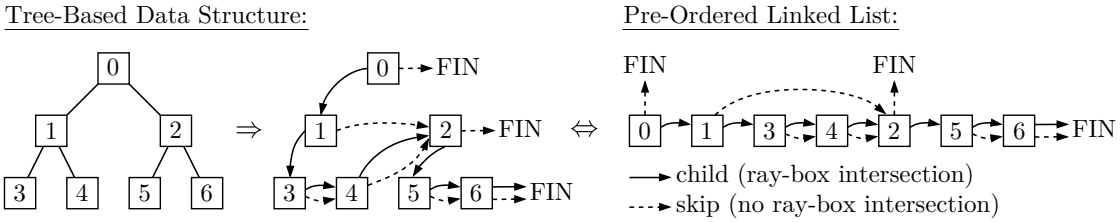

Figure 4.2.: Schematically illustration of translating a binary tree into a pre-ordered linked list. The image is from [WF12].

The idea is quite simple: each vertex of the tree has a pointer to one of its children, and a second pointer to its sibling vertex. With respect to ray-box intersection testing, the traversal is along a pointer to the child box (child-edge) if there is an intersection point, and otherwise it is along the pointer to the sibling vertex (skip-edge). Each vertex has out-degree 2 and in-degree ≥ 1, except the root. If there is no ray-box intersection, moving along the skip-edge allows to skip the entire subtree the current vertex would have been the root for. The traversal finishes if either the child-edge or the skip-edge brings us to a somehow distinguished vertex signaling the finish status ('FIN' in Fig. 4.2).

Although the linked list data structure is non-recursive, it remains unsuitable for the GPU for two reasons. First, moving along a skip-edge translates to an unordered access to the GPU's main memory. Second, moving along a skip-edge or a child-edge introduces branches in the execution of the active threads. If we use the GPU's processors such that every thread in a group (warp) executes the ray tracing program independently, branching leads to partial serialization of the execution within thread groups, resulting in significantly longer runtimes. Both of the two issues are in contrast to using the GPU in SIMD manner. However, we found using the GPU not in pure SIMD manner, but in MIMD manner, give notably performance improvements over a comparable vectorized and multithreaded CPU version of the ray tracer.

A point that was not addressed during the above mentioned software project course was work (re)distribution in the case of dynamic creation of secondary rays. Since secondary rays can extend the execution time of a thread, the entire thread group with that thread within has extended runtime. In the worst case the execution of the group is dominated by just one thread, and the group as a whole waits for one thread to finish. The same issue applies to thread blocks. If one thread group within a thread block has extended execution time due to secondary rays, the entire thread block cannot complete. If multiple thread blocks are affected by such irregular workloads, the GPU scheduler runs out of thread groups that can be scheduled. An important point to note is that each SIMD processor's thread scheduler(s) can manage up to 8 concurrent thread blocks only, so that in the worst case 8 thread groups in 8 different thread blocks have extended runtime, and instead of the maximum of 48 thread groups per processor only 8 are available for execution. The minimum number of thread blocks for the GPU scheduler(s) to use the GPU's compute units efficiently is about 20 per processor.

Xeon Phi and CPU Implementation

The implementation on the Xeon Phi and the CPU is almost identical to the GPU implementation. We also translate the octree into a pre-ordered linked list data structure, and we also use the processors in MIMD manner. Unlike the GPU, our implementation uses the Xeon Phi's and the CPU's SIMD units for a fast ray-triangle intersection test-

ing during the scene traversal. For that purpose the number of triangles per bounding box is adapted to be a multiple of the SIMD width on the Xeon Phi and the CPU, respectively—some boxes are filled up with dummy triangles.

The reason why we did not implement this SIMD traversal on the GPU is that for the programmer the GPU appears as a multi-core processor with 16×32 processor cores and with no SIMD units. For a typical GPU program that exploits data parallelism this view is correct as the program implicitly addresses SIMD parallelism. For a GPU program not written in data parallel manner it comes out that the GPU actually has just 16 physical processors (see Tab. 3.1). Fortunately, its interleaved multithreading concept allows to hide this circumstance to a certain extent (there are up to 768 logical processors). Further, there is always hope that not all threads in a thread group are serialized when not executing in SIMD manner. However, by using the GPU's SIMD processors in the same way as on the CPU, one would waste some performance as for a single ray only the ray-triangle intersection testing during the scene traversal provides enough potential for SIMD parallelism.

4.1.3. Ray Tracing and Load Balancing

With respect to the previous chapter, the determination of the color value of a pixel describes a multithreaded computation with the thread programs given by the 'MTC'-version of Listing 4.1. Rendering the entire image, there are as many independent multithreaded computations as there are pixels. For each pixel the computation starts with tracing the first-order ray associated with that pixel through the scene. If the ray hits a reflective/refractive triangle, secondary rays are created. The computational cost per pixel then increase.

Since the number of secondary rays per pixel is usually unknown, the best one can do is to partition the image into subimages and to distribute them across the processors available for execution (static processor-thread assignment). If the partitioning is fine enough—maybe at pixel level—, it should be unlikely that individual processors are given significantly more costly pixels than others, but it is not guaranteed. On the other hand, if we start up with a static assignment between processors and pixels (and hence threads), and if we allow processors that already finished their computations to acquire new work from processors that still execute, even workloads should be possible.

There are two essential criteria for such a load balancing scheme: It should be possible

1. to achieve at least the performance that can be obtained with an 'optimal' static assignment, and
2. to achieve a higher performance than the static assignment if the latter is not 'optimal'.

The evaluation of these two criteria is addressed in Sec. 4.1.4.

Subsequently, we use the work stealing scheme, as described in Chapter 3, except that for ray tracing there are no dependencies to be resolved. On the Xeon Phi and the CPU the non-blocking dequeue and the blocking dequeue, respectively, are utilized for thread sharing among the processors.

On the GPU, we use the non-blocking dequeue, too, but there are two additional challenges in adding dynamically created threads to the dequeue. Since threads within the same thread group do not execute independently of each other, but in wavefront manner, it does not make any sense to assign a dequeue to each processor that executes a thread. Rather we define a master thread for each thread group, and assign a per-thread-group dequeue to the processor that executes the master thread. While acquiring new threads for the group poses no difficulties, adding dynamically created threads to the thread group's dequeue is much more involved as there are up to 32 processors per thread group that may spawn multiple threads during their execution. A possible solution to this problem is as follows: since for ray tracing threads do not dependent on each other, the spawned thread is not distinguished from its parent thread, so that the execution order of the two is arbitrary. For each thread group we define a local buffer in the shared memory (low-latency on-chip memory that can be directly addressed) that contains up to twice the number of elements as there are threads per thread group. The elements that are stored in the local buffer are the child threads that may be spawned during the execution of the (parent) threads. The elements of the local buffer are accessible by the entire group. By putting the child threads instead of the parent threads into the local buffer, the thread group can continue executing the current wavefront without possible branches due to the depth-first execution of the child threads. In this way the recursive execution of the ray tracing function can be translated into an iterative wavefront execution. The size of the local buffer results from there are up to two spawns per thread in the case of refraction. For the wavefront execution to be efficient, the local buffer should hold one thread per group member in each step of the iteration. If the number of spawned threads is below the group size, additional threads can be acquired from the thread group's dequeue or the dequeue of other thread groups. Excess threads can be passed back to the thread group's dequeue. In our implementation for the GPU, we skip the last step, that is, we do not add threads to the dequeue.[1]

The procedure of filling up thread groups with threads is illustrated in Fig. 4.3, where 16 processors are considered. Figure 4.3 also assumes that there is at most one spawn per thread.

[1] The net effect of this approach is the same as adding threads to the dequeue, and then acquiring these threads in the next step of the execution.

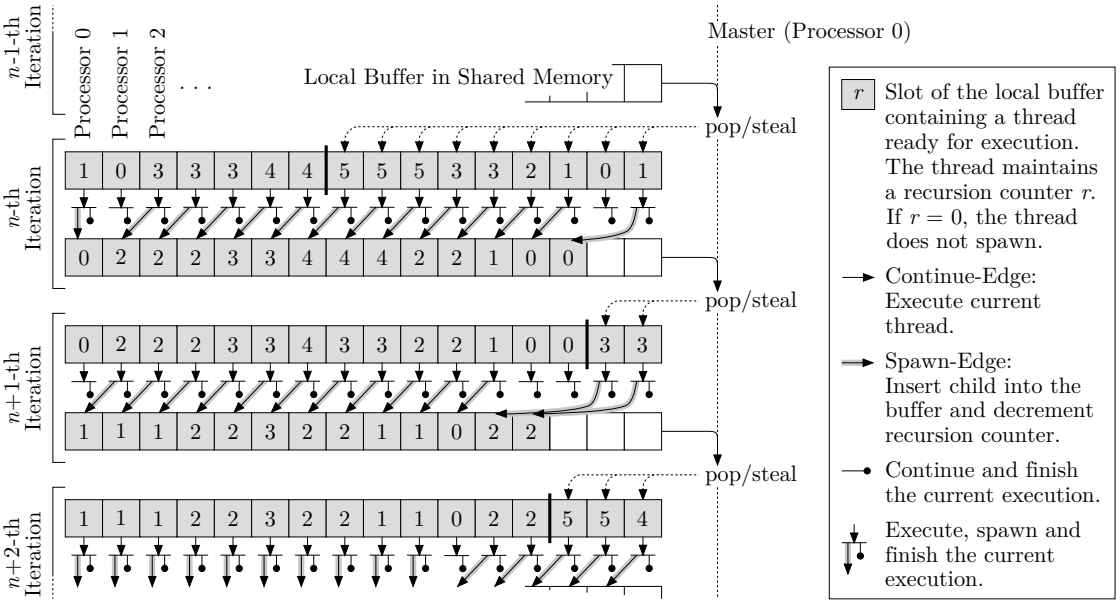

Figure 4.3.: Schematically illustration of filling up thread groups with threads during the execution of the ray tracing program. Thread groups here have size 16.

4.1.4. Performance Evaluation on GPU, Xeon Phi, and CPU

For the evaluation of the work stealing scheme we consider three different views on a scene that offer sufficient potential for dynamic thread creation during the rendering process. The scene that we use can be obtained from [dRu12]. The different view setups are summarized in Tab. 4.1.

For each setup we determined the work distribution per pixel by adding up the number

Scene: KingsTreasure [dRu12]
Triangles: 278230
Spot Light 1: RGB Color $(1.0, 0.9, 0.8)$, Position $(-40.0, 30.0, 40.0)$
Spot Light 2: RGB Color $(0.6, 0.4, 0.2)$, Position $(20.0, 20.0, -20.0)$
Ambient Light: RGB Color $(0.3, 0.3, 0.3)$

	Setup 1	Setup 2	Setup 3
Camera Position	$(2.0, 6.0, 16.0)$	$(14.0, 15.0, 20.0)$	$(14.0, 20.0, 20.0)$
Viewing Direction	$(-1.0, -0.5, -2.0)$	$(-3.0, -0.5, -4.0)$	$(4.0, -0.5, -4.0)$
Viewing Angle	$60°$	$60°$	$60°$
Max. Recursion Depth	8	8	8
Image Resolution	1024×1024 on Tesla M2090 and Xeon E5-2670 960×960 on Xeon Phi		

Table 4.1.: Benchmarking setups for rendering the KingsTreasure scene. The setups differ in the camera position and the viewing direction. For all devices the image resolution was chosen to be a multiple of the number of (logical) processors. On the Tesla M2090 we use 256 (logical) processors for the benchmark.

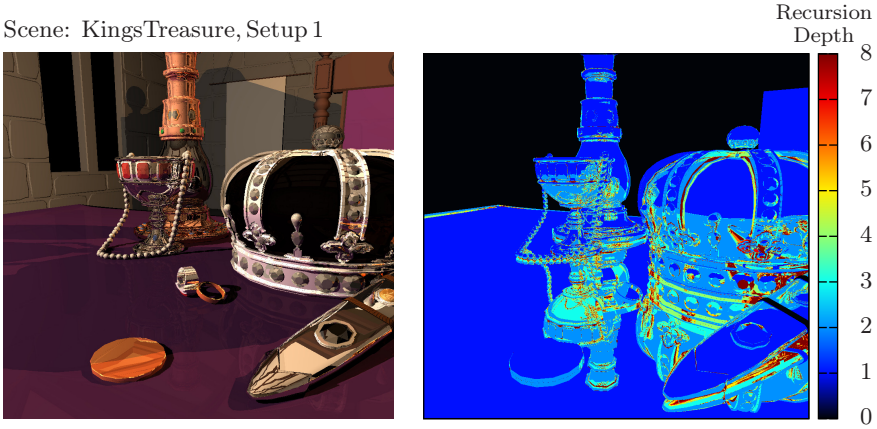

Figure 4.4.: Work distribution for rendering the KingsTreasure scene using setup 1. Color values towards 'red' correspond to a lot of work, whereas values towards 'blue' correspond to little work.

of recursive ray tracing function calls (see Listing 4.1). The distribution is shown in Fig. 4.4 for setup 1. It can be seen that some areas in the image are almost for free, whereas other ones are very costly. The distribution for setup 2 and 3 is illustrated in Appendix B.1, Fig. B.1.

Possible schemes for the partitioning of the image into subimages (tiles) are depicted in Fig. 4.5, where a 4-processor system is assumed. Obviously, the finer the grid the smaller the imbalance in the per-tile workloads. However, if the assignment of subimages to processors is static, the overall workload may differ from one processor to another.

We consider the following work distribution schemes:

Static Processor-Thread Assignment: The image is divided into subimages which then are assigned to a fixed number of processors. During the program run, all spawned threads are executed by their respective processors, resulting in no work sharing.

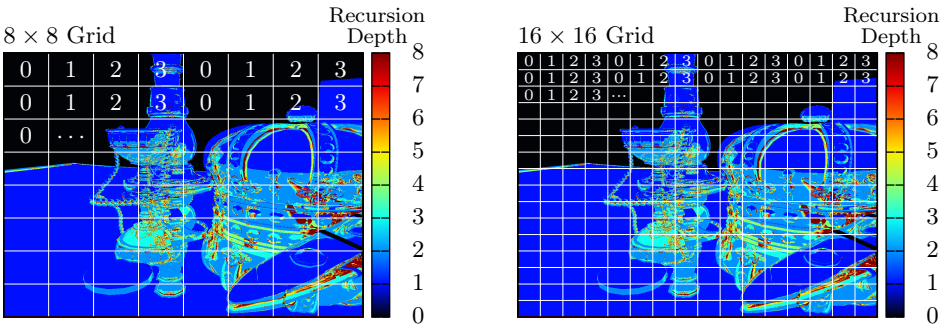

Figure 4.5.: Partitioning of the image of the KingsTreasure scene (setup 1) into an 8×8 and a 16×16 grid of subimages (tiles). Tiles are statically assigned to processors, where a 4-processor system is assumed. The number within the tiles refers to the ID of the processor the subimage is assigned to.

Processor-Thread Assignment using a Centralized Thread-Pool: The image is divided into subimages. The threads associated with these subimages are placed into a shared thread pool that is available to all processors. Processors (on the GPU the ones that execute the master threads) repeatedly acquire sets of threads (subimages) from the pool for execution. Our implementation of the pool is based on a counter variable that numbers the subimages, and that is atomically incremented by the processors when acquiring new work. Even for fine-grained grids, accessing the thread pool introduces almost no overhead as the execution of the threads within each subimage is more costly than incrementing the counter. Note: there is no work sharing when work once was acquired.

Processor-Thread Assignment using Work Stealing: The image is divided into subimages which are placed into the processors' dequeues (on the GPU the per-thread-group dequeues), resulting in a static assignment of work to processors. The (master) threads then execute the work stealing scheme.

CUDA-Managed Processor-Thread Assignment (GPU-only): The distribution of the subimages to the GPU's execution units is done by the GPU hardware scheduler. Principally, this scheme implements a centralized work pool in hardware. In this setting the GPU scheduler acts as an independent third party that assigns thread blocks and hence the subimages to the GPU's processors. Every time a thread block completes its execution, a new one, if there is any, is scheduled to the vacated processor.

The key difference to our centralized thread pool is in the granularity of the scheduling. While our scheme schedules subimages to thread groups, the GPU scheduler is only capable of scheduling subimages to groups of thread groups (thread blocks). A thread block completes its execution if and only if all thread groups within that block have finished their execution, potentially making some thread groups wait for others. Our scheme uses persistent threads which acquire new work by themselves The GPU scheduler only switches between the thread groups (interleaved multithreading), but none of the thread blocks is replaced by others, as all thread blocks finish their execution at almost the same time—this is when the computation as a whole is over.

On the Xeon Phi and the CPU we create as many persistent threads as there are logical processors (the persistent threads then encapsulate the execution of the threads that correspond to the subimages' pixels; see Sec. 3.3 for what is meant by 'persistent thread'); 240 on the Xeon Phi, and 16 on the CPU.

On the GPU we use only 256 of the possible 768 logical processors since our ray tracing kernel uses almost 3 times the amount of registers that is usually assigned to

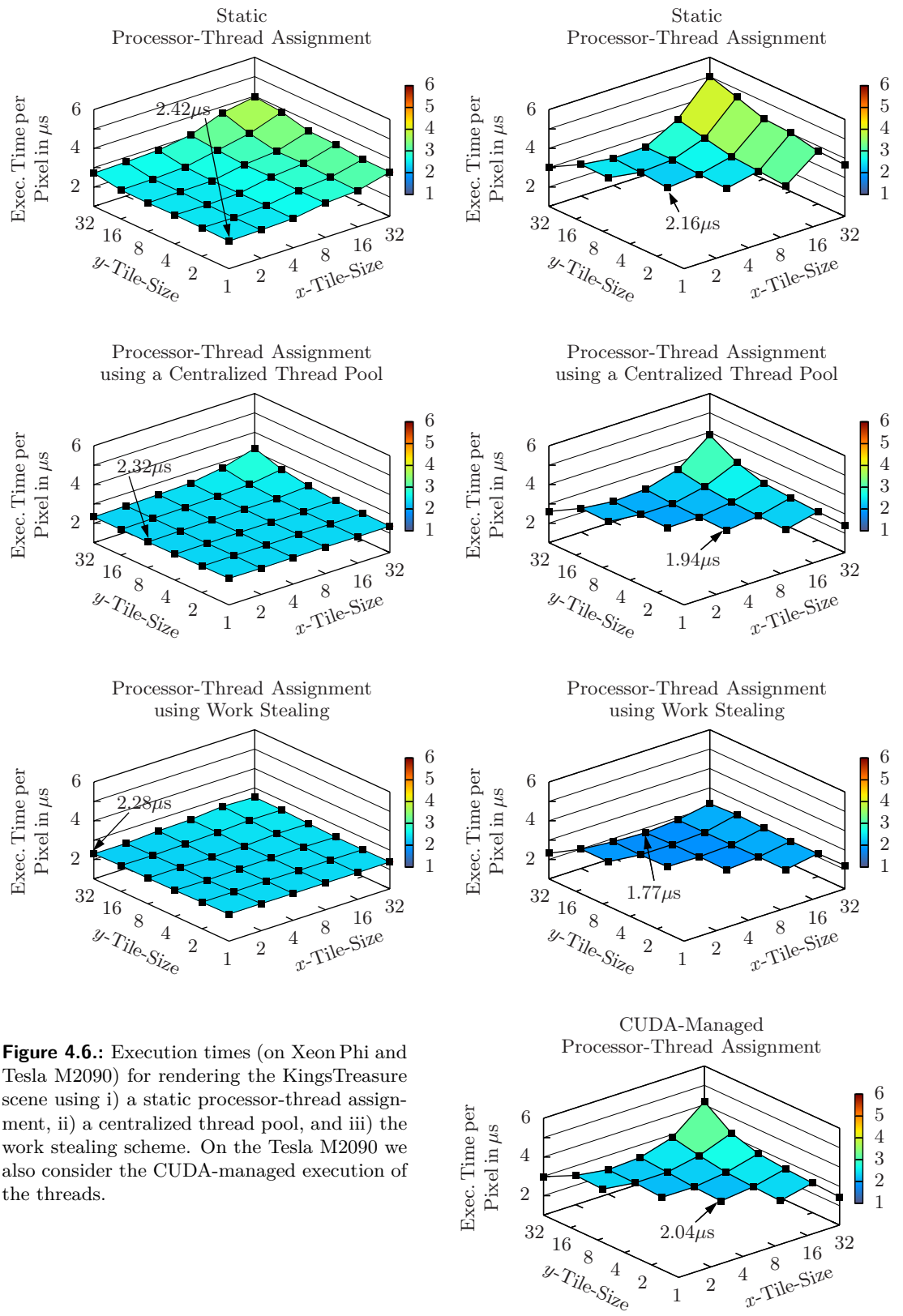

Figure 4.6.: Execution times (on Xeon Phi and Tesla M2090) for rendering the KingsTreasure scene using i) a static processor-thread assignment, ii) a centralized thread pool, and iii) the work stealing scheme. On the Tesla M2090 we also consider the CUDA-managed execution of the threads.

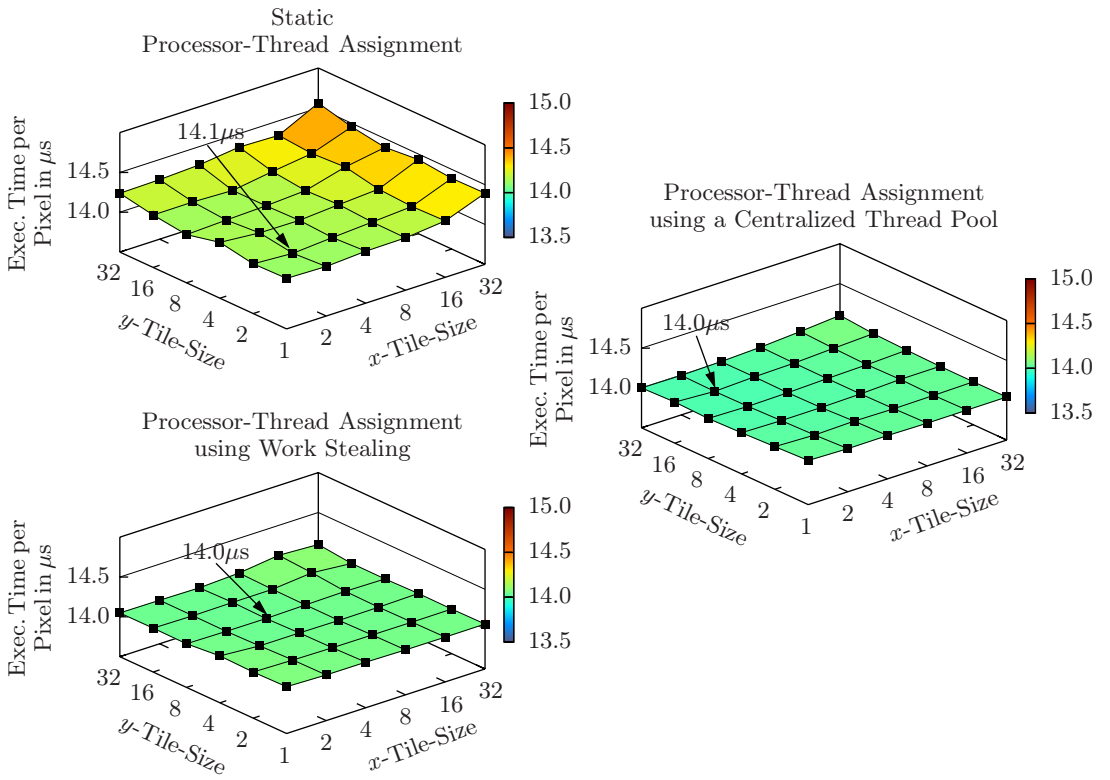

Figure 4.7.: Execution times (on the Xeon E5-2670) for rendering the KingsTreasure scene using i) a static processor-thread assignment, ii) a centralized thread pool, and iii) the work stealing scheme.

the processors. If we would use more than 256 logical processors the program would stall as the GPU scheduler then considers only a subset of the logical processors for the scheduling. The portion of the logical processors that are not considered would never execute as their execution can start if and only if other processors finish their execution—and these processors in turn finish their execution if and only if all work is done. This would cause a deadlock.

For each of the setups listed in Tab. 4.1, and for different tile sizes, rendering the image was done 6 times in succession (for statistics). The averaged execution times per pixel (in micro-seconds) are illustrated in Fig. 4.6 for the Xeon Phi and the Tesla M2090, and in Fig. 4.7 for the CPU. Error bars on the measurement data are not given as they are too small (so we simply excluded them). Fig. 4.8 and 4.9 illustrate the results for setup 2 and 3. Note that on the GPU the minimum tile size is given by the size of a thread group, which is 32 for the Tesla M2090.

For all setups it is recognizable that decreasing the tile size reduces the execution time per pixel, and hence the overall execution time for the image. Tile size reduction leads

to an implicit load balancing, since nearly the same amount of work is assigned to all processors on average. The larger the tiles the more potential for workload imbalances exists. Only in the case of the work stealing scheme imbalances can be compensated. The other schemes suffer from work once acquired cannot be redistributed to other processors if necessary. Since on the GPU there is also potential for irregularities in the workload on the level of the thread groups, due to the wavefront execution model, the GPU is particularly sensitive to workload imbalances. If we can avoid these imbalances the GPU performs good as can be seen from the execution times of the setups using the work stealing scheme. All setups confirm the work stealing scheme be superior to any other scheme in the tests.

On the CPU the implicit load balancing, due to the small number of processors (and the large number of tiles), results in all schemes perform almost equally well. For 32×32 subimages there are still 1024 tiles that are statically assigned to 16 processors, that is, 64 tiles per processor. On the GPU these 1024 tiles are distributed to 256 processors, so that 4 tiles are assigned to each of them. For the latter irregularities in the per-tile workload can affect the overall execution to a large extent. For that reason it should be obvious that the work stealing scheme on the CPU is less effective than on the GPU and the Xeon Phi. However, on the CPU the work stealing scheme lies at level with the centralized thread pool scheme so that using it does not slow down the execution.

When comparing execution times in Fig. 4.6 and Fig. 4.7, it can be seen that our CPU version of the ray tracer performs about a factor 6-8 below its GPU and Xeon Phi pendant, showing that in particular on the GPU the MIMD approach does not break down the performance. However, we assume that a SIMD version of the ray tracer would outperform our implementation.

Setup 3 was chosen as an example of a scene with almost no potential for irregularities in the workload (see Fig. B.1 for the work distribution of setup 3). The work stealing scheme performs on par with the other approaches for small tiles, and is superior to them when tile sizes become large.

The execution times for setup 3 suggest the assumption that the performance breakdown on the GPU, due to thread serialization within thread groups, is about a factor 2-3 for our ray tracing program. In setup 3 the linked list traversal should be similar for all threads so that the influence of the branching on the execution time can be neglected. According to Fig. 4.9, the performance of the ray tracer on the GPU is about three times the performance on the Xeon Phi. In setup 1 and 2, on the other hand, there is significantly more potential for thread serialization due to branches. For both of the two setups the GPU is faster than the Xeon Phi by about a factor 1.2 only. Thread serialization within thread groups thus seems to lower the performance on the GPU by about a factor 2-3.

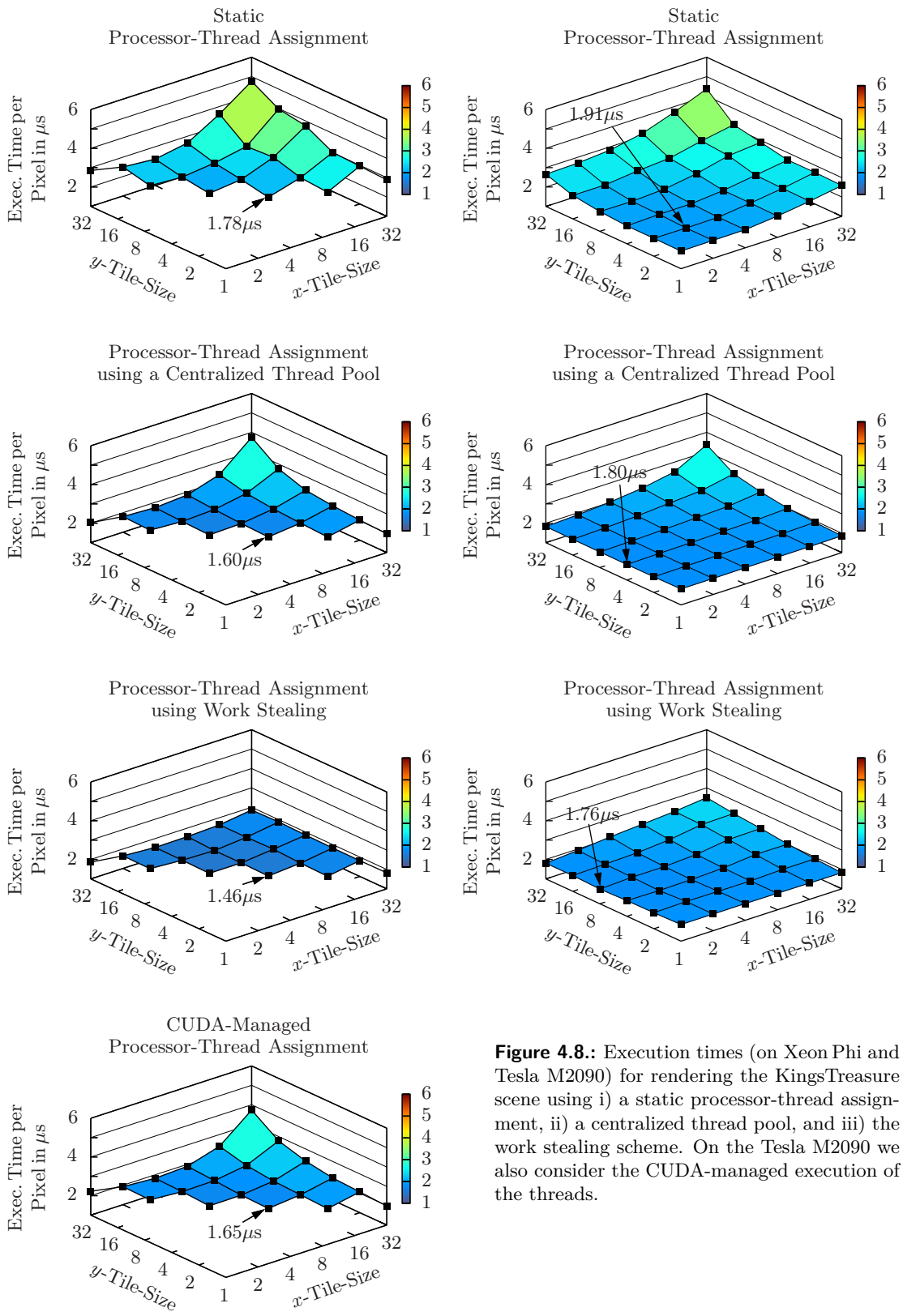

Figure 4.8.: Execution times (on Xeon Phi and Tesla M2090) for rendering the KingsTreasure scene using i) a static processor-thread assignment, ii) a centralized thread pool, and iii) the work stealing scheme. On the Tesla M2090 we also consider the CUDA-managed execution of the threads.

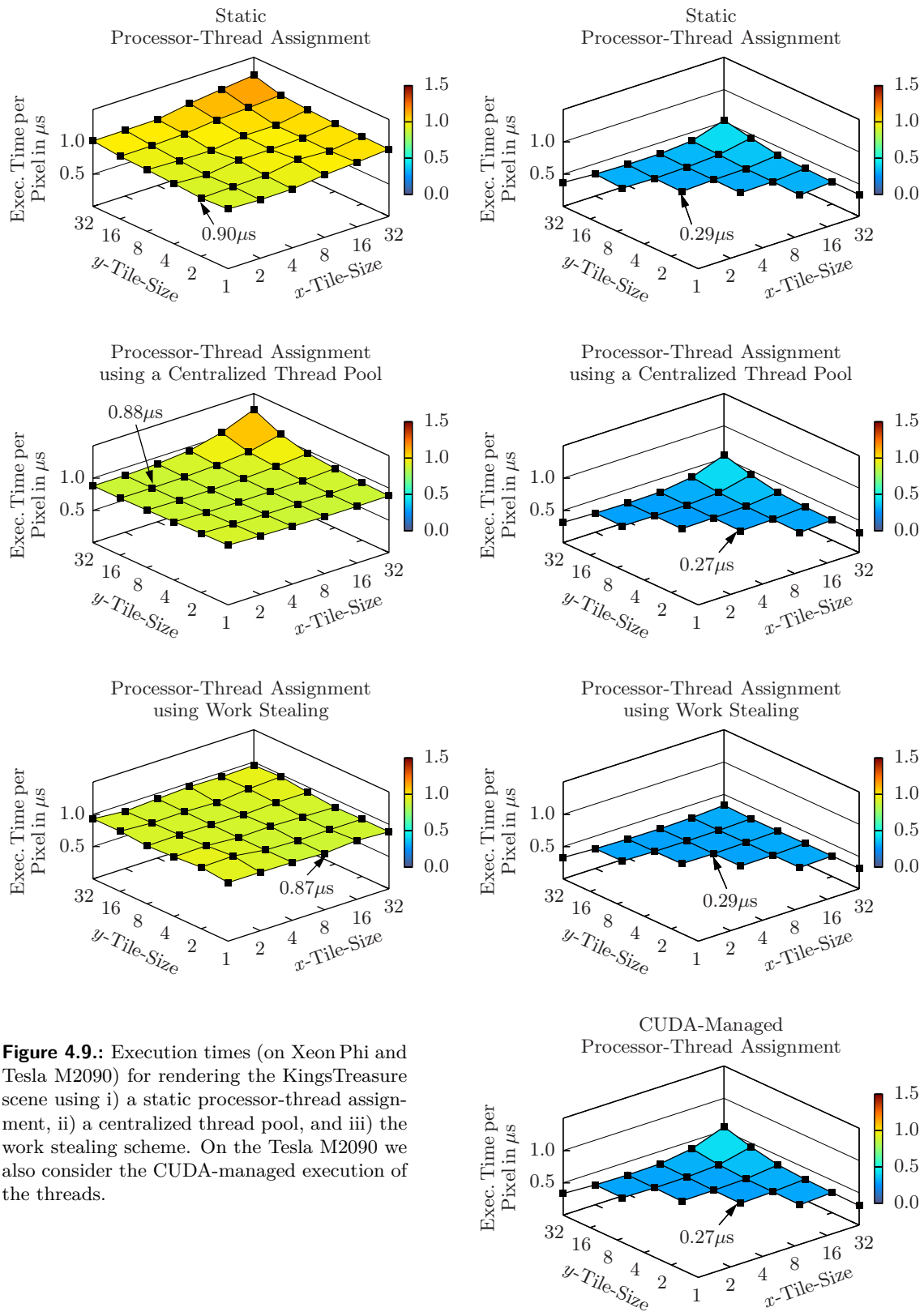

Figure 4.9.: Execution times (on Xeon Phi and Tesla M2090) for rendering the KingsTreasure scene using i) a static processor-thread assignment, ii) a centralized thread pool, and iii) the work stealing scheme. On the Tesla M2090 we also consider the CUDA-managed execution of the threads.

4.1.5. Validation of the Implementation

The implementations was validated by comparing images byte by byte. A program for that purpose can be found in `rayTracing/compareImages` on the CD. Images that were rendered using the same setup were identical across the different thread-processor assignment schemes considered in this section, except for work stealing. For all schemes other than work stealing the color values per pixel are summed up according to the depth-first execution of the threads, whereas for work stealing the order of the summation is the inverse. Since floating point addition is not associative, that is, $a + (b + c) \neq (a + b) + c$, images created with the work stealing scheme can be slightly different from those created using the other schemes.[2] However, we found all images created by different program runs (using the same setup) using the work stealing scheme be identical in every byte. We also did not found any visual deviations to images created using the other schemes.

4.2. Breadth-First Search

Breadth-first search (BFS) is a simple search algorithm in graph theory. Given a graph $G(E, V)$, where E is a set of edges and V a set of vertices, and a source vertex $s \in V$, the BFS algorithm systematically explores all vertices reachable from s in a *wavefront* manner. A vertex $u \in V$ is said to be reachable from s, if and only if there is a path $\langle v_0, v_1, \ldots, v_{k-1}, v_k \rangle$ with $(v_i, v_{i+1}) \in E$ for all $i \in \{0, 1, \ldots, k-1\}$, $v_0 = s$, and $v_k = u$. Each wavefront is associated with a distance $\delta \in \mathbb{N}_0$ to s. We define the $\delta = k$ wavefront $\mathcal{W}_k \subseteq V$ to contain all vertices $u \in V$ with $d[u] = k$, where $d[u]$ is the distance (fewest number of edges) from s to u. BFS starts with the wavefront $\mathcal{W}_0$ (containing the source vertex only), and then repeatedly explores $\mathcal{W}_{k+1}$ across the 'breadth' of $\mathcal{W}_k$ until all vertices reachable from s are discovered—hence the name breadth-first search. The exploration of $\mathcal{W}_{k+1}$ is along the edges $(u, v) \in E$ with $u \in \mathcal{W}_k$ and $v \in V$ adjacent to u. When discovered, vertices change their 'state' from *undiscovered* to *discovered*. Every vertex can be discovered at most once [CRL90].

During its execution BFS constructs a 'breadth-first tree' rooted at s. The BFS tree contains all vertices reachable from s. BFS also computes the *shortest paths* from s to all vertices $u \in V$ reachable from s—in terms of the fewest number of edges from s to u. The height of the BFS tree then corresponds to the diameter of the graph (the longest shortest path).

Adapted versions of the BFS algorithm are utilized in graph theory itself (Dijkstra's single-source shortest path (SSSP) algorithm, and connected component labeling, for

[2] As in parallel programming there is always potential for races when words are concurrently accessed by multiple processors, a modification of the code so that the order of the summation is the same for all schemes is not necessary in our opinion.

```
   function bfs(G(E,V),s) // Algorithm BFS
2:   for each u ∈ V do
       color[u]=WHITE     // Color of vertex u: WHITE (undiscovered)
       d[u]=INFINITY      // Distance to s: INFINITY ≙ 0xFFFFFFFF when 32-bit words are used
5:     p[u]=NULL          // Predecessor of u: NULL ≙ 0xFFFFFFFF when 32-bit words are used
6:   color[s]=GRAY        // Change color of source vertex s to GRAY (discovered)
     d[s]=0
8:   Q.enqueue(s)         // Previously, the queue was empty
9:   while Q ≠ ∅ do
       u=Q.head()        // Acquire head element of the queue, but do not remove
       for each v adjacent to u do
         if color[v]==WHITE then
           color[v]=GRAY // Change color of vertex v to GRAY (discovered)
           d[v]=d[u]+1
           p[v]=u
           Q.enqueue(v)
       Q.dequeue()       // Remove head element of the queue
18:    color[u]=BLACK    // Change color of vertex u to BLACK (discovered + done)
```

Listing 4.2: Breadth-first search algorithm (pseudo-code) [CRL90].

instance), circuit design, and cluster search algorithms in computational physics/chemistry, to name a few.

The BFS algorithm is illustrated in Listing 4.2 [CRL90], where

- Q refers to a (de)queue data structure, and
- vertex states are represented by colors:

  white (undiscovered),
  gray (discovered),
  black (discovered + done).

When vertices are discovered they change their color from white to gray. Vertices that belong to the current wavefront change their color from gray to black after all their adjacent vertices have been explored. As a consequence, gray vertices may have adjacent vertices that are white, whereas black vertices do not. The distinction between gray and black represents the progress in the wavefront exploration.

The execution of the BFS algorithm is illustrated in Fig. 4.10 using a random graph. The BFS algorithm applied to such graphs is at the core of cluster algorithms in statistical/computational physics. The simulation of the d-dimensional Ising model by means of the Swendsen-Wang multi-cluster algorithm [SW87], for instance, uses an intermediate representation of a spin system (a system made up of n spins placed onto n sites of a d-dimensional lattice), where spins that share a certain property and are in a somehow defined neighboring relation are organized into so-called 'clusters'. The clusters correspond to connected components in graph theory. The Swendsen-Wang cluster algorithm

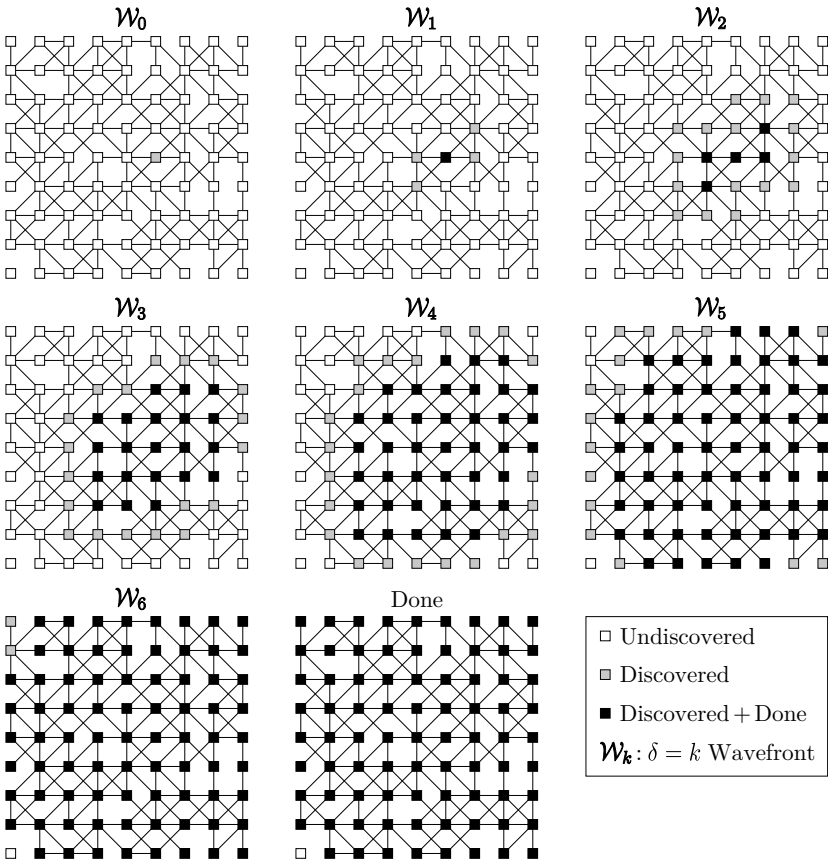

Figure 4.10.: Execution of the BFS algorithm using a random graph mapped onto a two-dimensional regular lattice. Colors correspond to the state of the vertices during the execution. Gray colored vertices all have the same distance to the source vertex (the gray vertex in the upper left subimage) and form a wavefront.

for the Ising model determines these clusters and applies modifications to them in order to move the spin system from one state to another one (within a Monte Carlo simulation). The clustering itself and the modifications of the clusters are done according to probabilities which incorporate simulation parameters.

Analysis

In Listing 4.2 algorithm BFS maintains three additional fields/arrays `color[]`, `d[]`, and `p[]` of size $|V|$ each. Since each vertex is discovered at most once, the queue also contains at most $|V|$ elements. Thus, the additional space S_{BFS} required by algorithm BFS is $S_{\text{BFS}}(G(E,V)) = \mathrm{O}(|V|)$.

The execution of lines 2 to 5 is in $\mathrm{O}(|V|)$. Lines 6 to 8 are $\mathrm{O}(1)$ operations each—all queue operations (`enqueue()`, `head()`, `dequeue()`) can be implemented as $\mathrm{O}(1)$ operations. The `while` loop from line 9 to 18 is executed $n \leq |V|$ times, because there

are at most $|V|$ vertices to discover. For every vertex u taken from the queue, all its adjacent vertices v, with $(u, v) \in E$, are considered for possible inclusion into the next wavefront. If we assume that an adjacency-list[3] is used, at most $\mathrm{O}(|E|)$ time is spend in scanning the list. Thus, the total running time $T_{\mathrm{BFS}}(G(E, V))$ of algorithm BFS is $T_{\mathrm{BFS}}(G(E, V)) = \mathrm{O}(|E| + |V|)$.

4.2.1. Parallel Implementation

In this subsection we focus on the parallelization of the BFS algorithm. With respect to multithreaded computations, the actual computation starts in line 9 in Listing 4.2. Threads correspond to the body of the `while` loop. For each vertex u taken from the queue, a thread program is executed with the number of spawns given by the number of undiscovered vertices v adjacent to u. Similar to ray tracing, thread programs can be executed independently. The only point in the execution where multiple concurrent threads might interfere with each other is the change of the color value of the vertices, possibly causing race conditions. How to solve this issue is described below in the text.

Other than ray tracing, the computation starts with just one processor being involved in the exploration of the $\mathcal{W}_1$ wavefront starting from $\mathcal{W}_0$ (the source vertex). Depending on the graph, wavefronts $\mathcal{W}_k$ relatively fast can contain a sufficiently large number of vertices so that more than one processor can be used for the parallel exploration of $\mathcal{W}_{k+1}$. Here, the work stealing scheme can serve as a mechanism for the redistribution of $\mathcal{W}_k$'s vertices across multiple processors at runtime. Each processor maintains two (de)queues. One of them contains the vertices of $\mathcal{W}_k$, whereas the other one contains the newly discovered vertices which in the next iteration form the wavefront $\mathcal{W}_{k+1}$. When finished all elements in all $\mathcal{W}_k$-(de)queues, the processors synchronize, change their (de)queues, and start with the exploration of $\mathcal{W}_{k+2}$ across the breadth of $\mathcal{W}_{k+1}$. The iteration continues until all vertices reachable from the source vertex are discovered.

Since thread executions are independent of each other, and also due to using two (de)queues for the current and the next-to-current wavefront, the data structure used for taking up vertices does not necessarily need to be a dequeue. For our implementation(s) we consider both queue and dequeue—therefore the (de)queue notation.

Listing 4.3 contains the pseudo-code for the parallel execution of the BFS algorithm—explanations are given subsequently.

[3]The information on whether two vertices are adjacent or not can be stored in an adjacency-matrix or an adjacency-list. The space requirements S for these representations are as follows: $S_{\mathrm{Matrix}}(E, V) = |V|^2$, $S_{\mathrm{List}}(E, V) = |V| + |E|$ for directed graphs, and $S_{\mathrm{List}}(E, V) = |V| + 2|E|$ for undirected graphs, respectively. If the graph is sparse, that is, if $|E| \ll |V|^2$, the adjacency-list is the preferred choice.

```
   function parallelBfs(G(E,V),s)
     PARALLEL for each u ∈ V do
     || color[u]=WHITE
     || d[u]=INFINITY
     || p[u]=NULL
     color[s]=GRAY
     d[s]=0
     Q[0].push(s)           // The source vertex is placed into the (de)queue of processor 0
     numVertices[2]={1,0}   // W_0 contains 1 vertex, and W_1 is empty so far
     finished=false         // There is work to do
     k=0                    // Start with δ = 0 wavefront W_0
12:  PARALLEL region:numProcessors=n
     ||  id=self()                                  // Get processor id: id∈{0,1,...,n−1}
14:  ||  while !finished do                         // As long as there is work to do continue
15:  ||    while numVertices[k%2]>0 do              // Current wavefront still contains vertices
     ||      if Q[id].empty() then                  // If own (de)queue is empty then
     ||        goto L1                              // go to label L1 and become a thief
     ||      if (u=Q[id].pop())!=NULL then          // Otherwise, try to acquire a vertex
19:  ||        for each v adjacent to u do          // and explore its adjacent vertices
20:  ||          if atomicCAS(&color[v],WHITE,GRAY) then
     ||            d[v]=d[u]+1   // This clause is executed if and only if the processor success-
     ||            p[v]=u        // fully changed v's color from white to gray. If so then push
     ||            Q[id].push(v) // v into the (de)queue and note that there is a further vertex
24:  ||            atomicINC(numVertices[(k+1)%2]) // in the next wavefront
     ||        color[u]=BLACK
26:  ||        atomicDEC(numVertices[k%2])          // Cancel u from current wavefront
27:  ||L1: while numVertices[k%2]>0 do               // Become a thief
     ||      victim='draw a random number uniform on {0,1,...,n−1}' // Select a victim processor
     ||      if (u=Q[victim].steal())!=NULL then
     ||        'execute the code between lines 19 and 26'
31:  ||    BARRIER                                  // Wait for all processors
     ||    if id==0 then                            // Processor 0 always exists
     ||      k++                                    // Move on to the next wavefront
     ||      if numVertices[k%2]==0 then            // If the current wavefront does not
35:  ||        finished=true                        // contain any vertices the search completes
36:  ||    Q[id].switch()                           // All processors change their (de)queues
37:  ||    BARRIER                                  // Wait for all processors
```

Listing 4.3: Parallel breadth-first search algorithm using work stealing (pseudo-code).

Termination and Break Conditions

The actual computation is in the parallel region from line 12 to 37. Here, n processors execute the BFS algorithm in wavefront manner in parallel. The exploration of $\mathcal{W}_{k+1}$ starting from $\mathcal{W}_k$ is between line 14 and 37. The array entry `numVertices[k%2]` holds the number of vertices in $\mathcal{W}_k$, and `numVertices[(k+1)%2]`[4] the number of

[4]The indices `k%2` and `(k+1)%2` map the current wavefront index k onto the two per-processor (de)queue(s) respecting the consecutive change of the two.

vertices in $\mathcal{W}_{k+1}$. Both of the two values match with the number of elements in the (de)queues associated with the respective wavefronts. For every element successfully removed from any of the $\mathcal{W}_k$-(de)queues, the value `numVertices[k%2]` is decremented by 1 atomically using the `atomicDEC()`[5] primitive. For every vertex that is discovered during the exploration, the value of `numVertices[(k+1)%2]` is incremented by 1 atomically using the `atomicINC()`[5] primitive. The respective lines are 24 and 26. All vertices of the current wavefront are done if and only if the value of `numVertices[k%2]` becomes zero. Then all processors leave their loop (line 15 for non-thief processors, and line 27 for thieves) and synchronize (line 31). Between line 31 and 37 processor 0 moves the wavefront one step forward and checks if the now-current wavefront contains any vertices. If not so, the variable `finished` is set to `true` (line 35). All processors then change their (de)queues (line 36), synchronize (line 37), and continue with the next wavefront unless `finished` is set to `true`. The execution then terminates—the termination is guaranteed as after a finite number of iterations, all reachable vertices are discovered and all (de)queues (and thus wavefronts) are empty.

Concurrent Change of Vertex Colors

The color of the vertices, when discovered, are changed from white to gray by means of the `atomicCAS()` primitive. Since the update of a certain vertex state is successful for only one out of $0 < n' \leq n$ processors, the clause between line 20 and 24 is executed exactly once for the respective vertex. As a consequence, every vertex v reachable from the source vertex s is discovered only once, and the vertex then is contained in one of the n (de)queues at any time until it becomes black.

The fact that every reachable vertex is discovered by exactly one processor (in the sense of changing its color from white to gray), and that it is acquired also by exactly one processor in the next iteration of the wavefront execution, guarantees that the `numVertices[]` entries alternately take on value zero. The inner loops in line 15 and 27 thus are guaranteed to terminate for every processor—the presence of the `BARRIER`s (line 31 and 37) enforce waiting for all n processors before starting the next iteration.

4.2.2. Performance Evaluation on Xeon Phi and CPU

For the evaluation of the performance of the parallel BFS algorithm, we generate a random graph that maps onto a three-dimensional periodic regular lattice with extents $L_1, L_2, L_3 \in \mathbb{N}$. Periodic means that lattice site (x_1, x_2, x_3) coincides with the site $(x_1 + n_1 L_1, x_2 + n_2 L_2, x_3 + n_3 L_3)$ for every (x_1, x_2, x_3) with $x_i \in \{0, 1, \ldots, L_i - 1\}$ and

[5] We use the `__sync_fetch_and_add(address,1)` and `__sync_fetch_and_sub(address,1)` compiler built-ins of the GNU and the Intel compiler for our x86 CPU and Xeon Phi implementation.

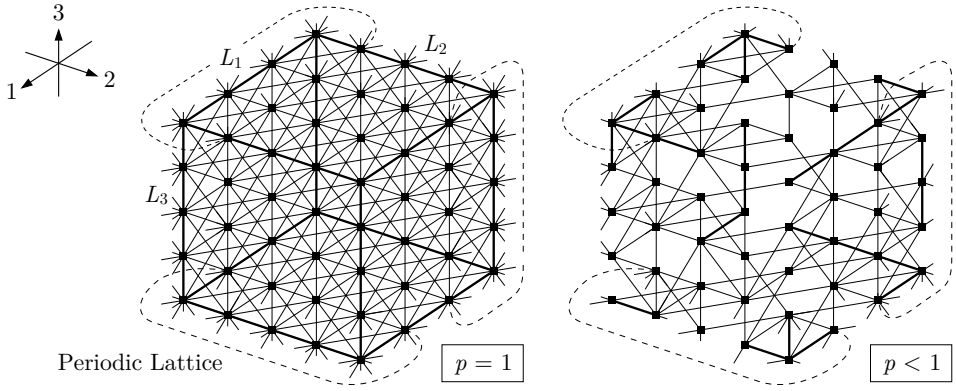

Figure 4.11.: Illustration of a random graph (mapped onto a three-dimensional periodic regular lattice) used for the evaluation of the parallel BFS algorithm. Vertices correspond to the lattice sites (x_1, x_2, x_3) with $x_i \in \{0, 1, \ldots, L_i - 1\}$ for $i = 1, 2, 3$. Edges are established with probability $0 \leq p \leq 1$.

$n_i \in \mathbb{Z}$ (for $i = 1, 2, 3$). The vertices of the graph correspond to lattice sites, and edges are established between (x_1, x_2, x_3) and $(x_1 + \Delta_1, x_2 + \Delta_2, x_3 + \Delta_3)_{\Delta_{1,2,3}=-1,0,1} \neq (x_1, x_2, x_3)$ with probability $0 \leq p \leq 1$. Thus, the graph $G(E, V)$ has $|V| = L_1 \times L_2 \times L_3$ vertices and $0 \leq |E| \leq 13|V|$ edges. If $p = 1$, the graph is 26-regular. A possible graph is illustrated in Fig. 4.11 (such graphs can be found in computational physics/chemistry).

For the graph representation we use an adjacency-list.[6] Its implementation uses the `vector<T>` data type of the C++ standard template library (STL). Vertices are represented as a C/C++ struct/record containing the vertex itself and a list of its adjacent vertices (see Listing 4.4). The latter is sorted in ascending order to reduce unfortunate memory accesses when scanning the adjacency-list.

```
typedef struct{
  int id;
  std::vector<int> adj; // List of adjacent vertices
} vertex;

class graph{
 public:
  // Constructor
  graph(int n,float p){
    'create random graph with n vertices and with probability p for edge creation'
  }
  // Attributes: List of vertices accessable through vertices.at(v.id), for instance
  std::vector<vertex> vertices;
};
```

Listing 4.4: Graph representation (pseudo-code).

[6] In our implementation we do not use any special kind of graph representation so that it should be possible to easily import kinds of graphs other than the one chosen here. We therefore assume that there is no loss of generality with the kind of graph chosen for the evaluation of the BFS algorithm.

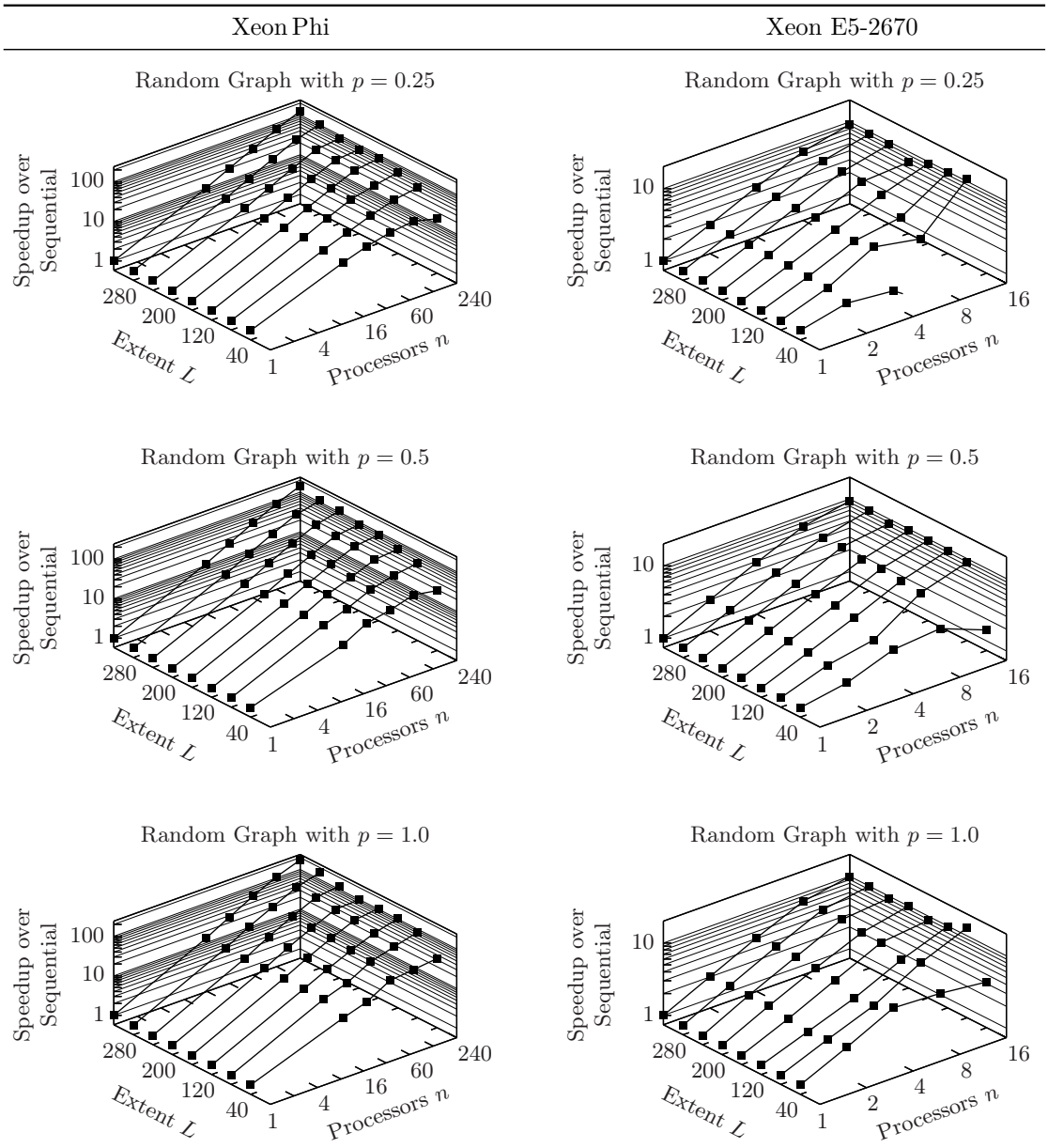

Figure 4.12.: Speedup of the parallel BFS algorithm over sequential BFS on Xeon Phi and CPU for a random graph with $|V| = L \times L \times L$ vertices, and $0 \leq |E| \leq 13|V|$ edges established with probability $p \in \{0.25, 0.5, 1.0\}$.

The parallel region in Listing 4.3 is realized by means of OpenMP. OpenMP threads correspond to persistent threads introduced in the previous sections/chapters, and are assigned to (logical) processors. In particular they encapsulate the operations between line 12 and 37 in Listing 4.3. Barriers are also realized by means of OpenMP (`#pragma omp barrier`).

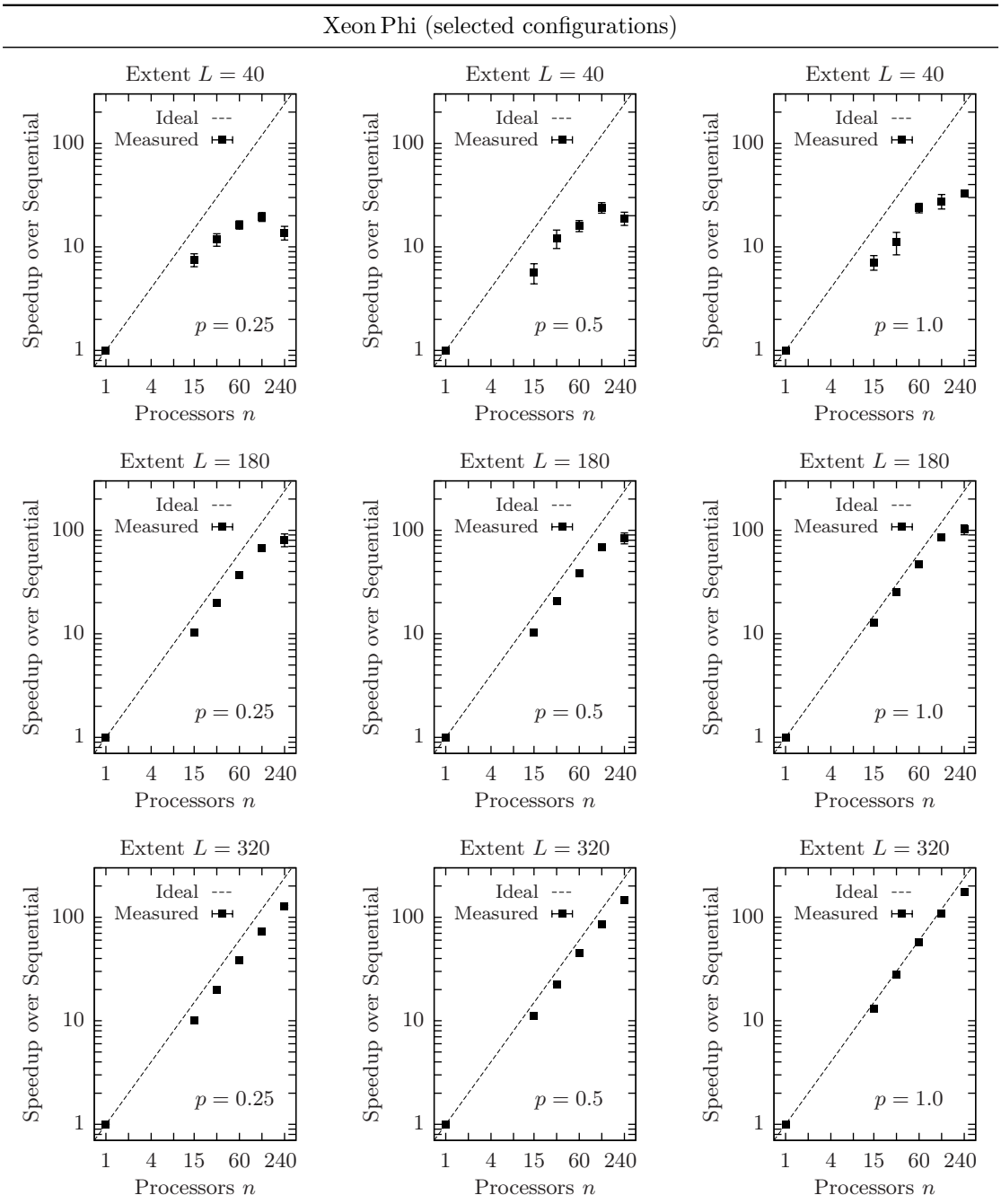

Figure 4.13.: Speedup of the parallel BFS algorithm over sequential BFS on Xeon Phi for a random graph with $|V| = L \times L \times L$ vertices, and $0 \leq |E| \leq 13|V|$ edges established with probability $p \in \{0.25, 0.5, 1.0\}$.

On the Xeon Phi the non-blocking dequeue data structure is used (see Sec. 3.4.2, and Appendix A.2), whereas on the CPU we use a blocking queue. Vertices of the current wavefront are acquired chunk-wise with the chunk size being equal to 8 on the Xeon Phi

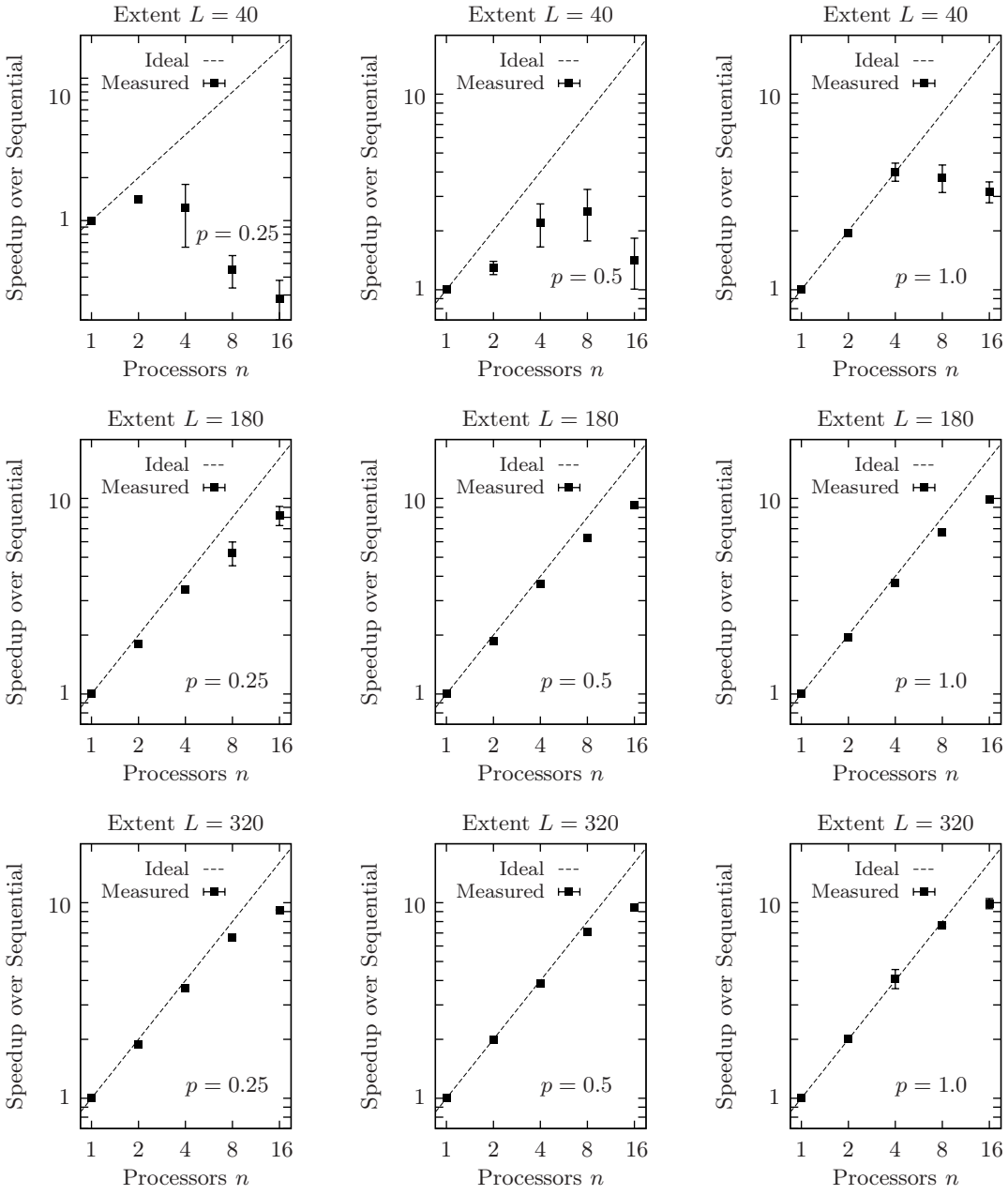

Figure 4.14.: Speedup of the parallel BFS algorithm over sequential BFS on CPU for a random graph with $|V| = L \times L \times L$ vertices, and $0 \leq |E| \leq 13|V|$ edges established with probability $p \in \{0.25, 0.5, 1.0\}$.

and 32 on the CPU—with these values we achieved the best performance. Similar to ray tracing the number n of processors used for the execution is varied over a meaningful range matching the number P of processors on the device (see Tab. 3.1). Benchmarks

	Xeon Phi			Xeon E5-2670		
	n	Exec. Time [s]	Speedup	n	Exec. Time [s]	Speedup
$L = 40$	1	0.220(3)	1.0	1	0.015(8)	1.0
	15	0.031(5)	7.1(1.1)	2	0.00781(19)	1.9(1.0)
	30	0.020(5)	11(3)	4	0.0038(4)	4(2)
	60	0.0092(10)	24(3)	8	0.0041(7)	4(2)
	120	0.0080(13)	28(5)	16	0.0048(6)	3(2)
	240	0.0068(4)	32(19)			
$L = 180$	1	20.9(3)	1.0	1	1.180(3)	1.0
	15	1.642(10)	12.73(19)	2	0.603(5)	1.957(17)
	30	0.824(3)	25.4(4)	4	0.319(8)	3.70(9)
	60	0.4425(9)	47.2(7)	8	0.177(7)	6.7(3)
	120	0.2461(18)	85(6)	16	0.11922(11)	9.90(3)
	240	0.21(2)	100(10)			
$L = 320$	1	151.0(6)	1.0	1	7.91(5)	1.0
	15	11.52(6)	13.11(9)	2	3.91(2)	2.023(16)
	30	5.462(17)	27.65(14)	4	1.93(22)	4.1(5)
	60	2.650(6)	57.0(3)	8	1.033(17)	7.66(13)
	120	1.388(8)	108.8(8)	16	0.80(5)	9.9(6)
	240	0.87(5)	174(10)			

Table 4.2.: Execution times of the BFS algorithm for a random graph with $L \times L \times L$ vertices and edges established with probability $p = 1.0$. Statistical errors are given in brackets: 6.7(3) means 6.7 ± 0.3; 1.957(17) means 1.957 ± 0.017.

were done for different sized graphs with $L \times L \times L$ vertices and $L \in \{40, 80, \ldots, 320\}$, and with edges established with probability $p \in \{0.25, 0.5, 1.0\}$. The source vertex is chosen at random, and for each such configuration the execution is repeated 6 times in succession for statistics.[7] For the speedup calculation the 1-processor execution uses an optimized BFS code without the overhead (atomic operations and (de)queue operations) of the parallelized version. The benchmarking results are illustrated in Fig. 4.12 - 4.14.

As can be seen from these images, for 'large' graphs and p not too small, the performance gain over the 1-processor execution increases almost linearly with the number n of processors. For 'small' graphs and/or 'small' p, the size of the wavefronts decreases and hence the potential for massive parallelism, as shown in the $L = 40$ images in Fig. 4.13 and Fig. 4.14.

Table. 4.2 lists the $p = 1.0$ execution times the speedups in Fig. 4.13 and Fig. 4.14 are calculated from. For 'large' graphs with $p = 1.0$ the parallel BFS algorithm completes after almost the same amount of time on both the Xeon Phi and the CPU when all

[7] For $L = 40$ on the CPU we averaged over 20 executions for reasonable measurement data.

logical processors are used. The reason why the Xeon Phi lies at level with the CPU is that the code is not vectorized. As the Xeon Phi draws its compute performance from a combination of massive parallelism and vectorization, it is at a disadvantage compared to the CPU which here can come up with a strong per-core compute performance even without vectorization. A vectorized version of the code might perform twice as fast on the Xeon Phi as on the CPU due to the 'Phi's 512-bit vector units—current x86 CPUs have 128-bit or 256-bit vector units. However, the measurement data confirm the functioning of the work stealing scheme for the BFS algorithm.

4.2.3. Validation of the Implementation

The implementation was validated as follows: For any source vertex $s \mathrel{\hat{=}} (x_1, x_2, x_3)$ with $x_i \in \{0, 1, \ldots, L_i - 1\}$ (for $i = 1, 2, 3$), and for $p = 1.0$, the diameter of the graph is $\lfloor \max\{L_1, L_2, L_3\}/2 \rfloor$. This value could be confirmed for different s and L_i values. For $p = 1.0$ we also checked that all vertices of the graph have been actually discovered after the execution of BFS. We further compared the entries of the array `d[]` (containing the distances to s) produced by the non-parallelized BFS code with entries produced by the parallelized version. For different values of p and L_i we found them be the same for all vertices, that is, both implementations produce the same `d[]` array.

4.2.4. Scalable Work Stealing & State of the Art

Although using the work stealing scheme for dynamic work (re)distribution during the execution of the BFS algorithm seems to result in an acceptable utilization of the underlying parallel computer hardware, our approach is primarily designed for shared memory machines. As (really) large graphs (usually) do not fit into the main memory of a shared memory system, our implementation needs to be adjusted to run on cluster computers or even supercomputers—the work stealing scheme as presented in this thesis also needs to be adapted. In [DLS+09] the scalability of the work stealing scheme is investigated. The authors found randomized work stealing to also work for cluster computers with thousands of processors—their benchmarks include unbalanced tree search.

Another approach to massively parallel tree search on cluster computers (and supercomputers) is described by T. Schütt, A. Reinefeld, and R. Maier [SRM13]. The authors utilize the MapReduce programming model—inspired by the 'map' and 'reduce' functions commonly used in functional programming. Their approach is to put the parallel wavefront exploration into the map-phase, and then to (re)distribute newly created work over the processors of the system in the reduce-phase, to put it simple. The scalability of MR-search, as referred to by the authors, is shown in their paper.

5. Summary & Conclusion

In this thesis we evaluated dynamic load balancing schemes on (massively) parallel computer architectures with the focus on work stealing. After having introduced the theoretical background of multithreaded computations in Chapter 2, we moved on to dynamic load balancing. Following the work of R. D. Blumofe and Ch. E. Leiserson [BL99], and D. Cederman and P. Tsigas [CT08], we introduced greedy schedules, and took up a randomized work stealing algorithm for fully-strict multithreaded computations.

In Chapter 3 we introduced the computer hardware used in this thesis: Nvidia Tesla M2090 GPU, Intel Xeon Phi, and Intel Xeon E5-2670 x86 octa-core CPU. In particular we briefly described Nvidia's Fermi GPU architecture, and Intel's Many Integrated Core (MIC) architecture. We gave implementations of multithreaded data structures—'lock' and (non-)blocking 'dequeue', namely—for the realization of load balancing on these device. On the GPU we found non-blocking implementations be essential for good performance and scalability. On the Xeon Phi and the CPU also blocking versions of these data structures give acceptable performance. Our results on the GPU are compatible with results presented in [CT08]. We were also concerned with the implementation of the randomized work stealing algorithm described in Chapter 2. We described an approach for dependency resolution, and we demonstrated the functioning of a respective implementation on both Xeon Phi and GPU. A direct comparison with the Cilk programming language was somehow confusing as the Cilk version of our code did not show the expected scaling behavior. It is still an open question for us whether we made unfortunate use of Cilk, or whether it is on Cilk itself.

In Chapter 4 we put the work stealing scheme into practice by applying it to real-world applications—ray tracing and breadth-first search (BFS). For both kinds of applications we were able to demonstrate the suitability of the work stealing scheme. While our ray tracer on the GPU and the Xeon Phi gives measurable performance gains over a parallelized CPU implementation (also using work stealing), in the case of the BFS algorithm the Xeon Phi and the CPU lie at level (for sufficiently 'large' graphs) with respect to program execution times—our parallel implementation of the BFS algorithm does not make use of vector operations, resulting in the Xeon Phi performs below its limits.

For our GPU ray tracing code using work stealing we additionally introduced an effective mechanism of work (re)distribution within thread groups described in Sec. 4.1.3. Since GPUs are commonly used for the execution of massively parallel SIMD programs, the execution of non-SIMD programs on the GPU suffers from partial serialization of these programs on the level of the GPU's 'vector units' thread groups are mapped onto dynamically at runtime. Our scheme helps reducing the effects of the serialization within thread groups in the context of multithreaded computations with dynamic work creation. In particular, the topic of handling thread serialization within thread groups, when computations become irregular, can serve as a starting point for further investigations in the field of GPU programming.

Due to the timeout of this thesis, we did not give an implementation of the parallel BFS algorithm using the GPU. The paper of [LWH10] here might serve as a basis for a GPU implementation.

Also the vectorization of the BFS algorithm for a fast execution on the Xeon Phi seems to be possible with its gather and scatter vector operations—these operations will be also available with the AVX 2 vector extension of upcoming CPU generations.

In Summary we found the work stealing scheme be suitable for the kinds of applications considered in this thesis. With respect to its implementation, the most challenging part was on the design of the shared data structures that are at the core of work stealing. Especially on the GPU, writing PTX code (similar to inline assembly code) for cache-volatile memory loads/stores was necessary to make the concurrent data structures do not stuck at any time. Merging the work stealing scheme with the ray tracer and the BFS code, respectively, then was quite easy.

Since the integration of the work stealing scheme requires only little modification of the data structures, we assume that a much wider class of applications with irregular workloads can benefit from it.

On the other hand, we found the work stealing scheme perform well only if the task sizes are sufficiently large so that the overhead of the scheduling can be neglected. For the ray tracing procedure and the BFS algorithm this was the case. For applications with a much finer granularity of the parallelism and/or with small-sized tasks work stealing might not be the preferred choice. Especially on large-scale cluster computers the communication overhead for the work (re)distribution is much more costly than on shared memory machines (as considered in this thesis). Although in [DLS+09] the suitability of the work stealing scheme for irregular computations on cluster computers is demonstrated, however, state of the art techniques like MapReduce might be more appropriate.

A. Blocking Dequeue, Non-Blocking Dequeue, Load Balancing

In this chapter we give concrete implementations of the blocking and non-blocking dequeue for Nvidia GPUs of compute capability at least 1.1.

A.1. Blocking Dequeue (GPU)

```
////////////////////////////////////////////////////////
// blockingDequeue.cuh
////////////////////////////////////////////////////////
#include <stdint.h>

// Implementation of the lock
__device__ void lock(uint32_t *lock){while(atomicCAS(lock,0,1)){;}}
__device__ bool tryLock(uint32_t *lock){return !atomicCAS(lock,0,1);}
__device__ void unlock(uint32_t *lock){atomicExch(lock,0);}

// A simple thread definition
typedef struct {uint32_t x;} thread;
__host__ __device__ thread makeThread(const uint32_t x=0){
  thread temp;
  temp.x=x;
  return temp;}

class dequeue{
 public:
  // The dequeue is initialized by the host
  __host__ dequeue(const uint32_t size = 0){dq=NULL;init(size);}

  __host__ void init(const uint32_t size){
    if(dq!=NULL||size==0||size>0xFFFFFFFE)
      return;
    cudaMalloc((void **)&dq,size*sizeof(thread));
    this->size=size;tail=0;head=0;lockVar=0;}
```

```
// The dequeue is deleted by the host
__host__ ~dequeue(){
  if(dq!=NULL)
    cudaFree(dq);
  dq=NULL;}

// The host may insert threads into the dequeue
__host__ void insert(const thread *t=NULL,const uint32_t numThreads=0){
  if(dq==NULL||t==NULL||numThreads==0||numThreads>size)
    return;
  cudaMemcpy(dq,t,numThreads*sizeof(thread),cudaMemcpyHostToDevice);
  tail=numThreads;}

__host__ __device__ bool empty(){return head==tail;}

__host__ __device__ bool full(){return tail==size&&tail>head;}

__device__ bool push(const thread t){
  if(dq==NULL||tail==size)
    return false;
  lock(&lockVar);
  dq[tail++]=t;
  unlock(&lockVar);
  return true;}

__device__ bool steal(thread *t=NULL){
  if(dq==NULL||t==NULL||tail==head)
    return false;
  bool success=false;
  if(tryLock(&lockVar)){
    if(head<tail){
      (*t)=dq[head++];
      success=true;}
    unlock(&lockVar);}
  return success;}

__device__ uint32_t stealChunk(thread *t=NULL,const uint32_t chkSize=1){
  if(dq==NULL||t==NULL||tail==head)
    return 0;
  uint32_t threadsStolen=0;
  if(tryLock(&lockVar)){
    if(head<tail){
      threadsStolen=(tail-head)<chkSize?tail-head:chkSize;
      for(uint32_t i=0;i<threadsStolen;i++)
        t[i]=dq[head+i];
```

```
        head+=threadsStolen;}
      unlock(&lockVar);}
    return threadsStolen;}

  __device__ bool pop(thread *t=NULL){
    if(dq==NULL||tail==0)
      return false;
    bool success=false;
    lock(&lockVar);
    if(head<tail){
      (*t)=q[--tail];
      success=true;
      if(tail==head){
        tail=0;head=0;}}
    unlock(&lockVar);
    return success;}

 private:
  uint32_t size,lockVar;
  volatile uint32_t head,tail;
  thread *dq;
};
```

Listing A.1: Blocking dequeue implementation for Nvidia GPUs of compute capability at least 1.1.

For the source code, see Sec. A.3.

A.2. Non-Blocking Dequeue (GPU)—Extended Version

This implementation of the non-blocking dequeue for Nvidia GPUs of compute capability at least 1.1 is based on Listing 3.4. It extends the said implementation by the `popWarp()` and `stealWarp()` operations which allow to acquire sets of elements of the warp size (32 on the Tesla M2090). Further, this implementation is not restricted to contain at most 65535 elements, but it allows for $2^{24} - 1$ elements.

We use inline PTX code (Parallel Thread Execution; assembly like programming language) to explicitly make reads and writes from/to critical variables be cache-volatile. We do not use the `-Xptxas -dlcm=cv` compile option since the CUDA 4.0 and 4.1 compiler somehow produces code that is significantly slower than the one given in Listing A.2. Our implementation should run with CUDA 4.0, 4.1, and 5.0.

The implementation was tested by means of the test program `testDequeue` in `threadsafeDataStructures/dequeue/gpu` on the CD.

For the source code, see Sec. A.3.

```
////////////////////////////////////////////////////
// nonBlockingDequeue.cuh
////////////////////////////////////////////////////
#include <stdint.h>

// Cache volatile read
__device__ uint32_t volatileRead(uint32_t *address){
  uint32_t temp;
  __asm__ __volatile__("ld.volatile.global.u32 %0,[%1];"
                       :"=r"(temp):"l"(address));return temp;}

// Write through the cache
__device__ void volatileWrite(uint32_t *address,const uint32_t value){
  __asm__ __volatile__("st.volatile.global.u32 [%0],%1;"
                       ::"l"(address),"r"(value));}

// Dequeue index: 48 bits for the dequeue, 8 bits to address the ABA problem
typedef union {uint8_t u8[4];uint32_t u32;} queueIdx;

// A simple thread definition
typedef struct{uint32_t x;} thread;
__host__ __device__ thread makeThread(const uint32_t x=0){
  thread temp;temp.x=x;
  return temp;}

#define WARP_SIZE (32)
#define MIN(X,Y) (X<Y?X:Y) // Meaning: if X<Y then X else Y

class dequeue {
 public:
  // The dequeue is initialized by the host
  __host__ dequeue(const uint32_t size=0){dq=NULL;init(size);}

  __host__ void init(const uint32_t size){
    if(dq!=NULL||size==0||size>0xFFFFFF) // Maximum number of elements is 2^24-1
      return;
    cudaMalloc((void **)&dq,size*sizeof(thread));
    this->size=size;tail=0;head.u32=0;}

  // The dequeue is deleted by the host
  __host__ ~dequeue(){
    if(dq!=NULL)
      cudaFree(dq);
    dq=NULL;}
```

```
// The host may insert threads into the dequeue
__host__ void insert(const thread *t=NULL,const uint32_t numThreads=0){
  if(dq==NULL||t==NULL||numThreads==0||numThreads>size)
    return;
  cudaMemcpy(dq,t,numThreads*sizeof(thread),cudaMemcpyHostToDevice);
  tail=numThreads;}

// Logical AND with 0xFFFFFF extracts the lower 24 bits
__device__ bool empty(){return ((head.u32)&0xFFFFFF)>=tail;}

__device__ bool full(){return tail==size&&tail>((head.u32)&0xFFFFFF);}

__device__ bool push(const thread t){ // See Listing 3.4
  if(dq==NULL)
    return false;
  uint32_t oldTail;
  oldTail=volatileRead(&tail);
  if(oldTail==size)
    return false;
  dq[oldTail++]=t;
  volatileWrite(&tail,oldTail);
  return true;}

__device__ bool steal(thread *t=NULL){ // See Listing 3.4
  if(dq==NULL||t==NULL)
    return false;
  queueIdx oldHead,newHead;
  uint32_t oldTail;
  oldTail=volatileRead(&tail);
  oldHead.u32=volatileRead(&head.u32);
  if(oldTail<=((oldHead.u32)&0xFFFFFF))
    return false;
  (*t)=dq[(oldHead.u32)&0xFFFFFF];
  newHead=oldHead;
  newHead.u32++;
  if(atomicCAS(&head.u32,oldHead.u32,newHead.u32)==oldHead.u32)
    return true;
  return false;}

__device__ uint32_t stealWarp(thread **tailPtr=NULL){
  if(dq==NULL||tailPtr==NULL)
    return 0;
  queueIdx oldHead,newHead;
  uint32_t oldTail;
  (*tailPtr)=NULL;
  oldTail=volatileRead(&tail);
```

```
    oldHead.u32=volatileRead(&head.u32);
    // The last 2*WARP_SIZE elements are left to the owner
    if(oldTail<=(((oldHead.u32)&0xFFFFFF)+2*WARP_SIZE))
      return 0;
    // The thief gets a pointer to the dequeue position from where elements can be taken
    (*tailPtr)=&dq[(oldHead.u32)&0xFFFFFF];
    newHead=oldHead;
    newHead.u32+=WARP_SIZE;
    if(atomicCAS(&head.u32,oldHead.u32,newHead.u32)==oldHead.u32)
      return WARP_SIZE;
    (*tailPtr)=NULL;
    return 0;}

  __device__ bool pop(thread *t=NULL){ // See Listing 3.4
    if(dq==NULL||t==NULL)
      return false;
    queueIdx oldHead,newHead;
    uint32_t oldTail;
    oldTail=volatileRead(&tail);
    if(oldTail==0)
      return false;
    volatileWrite(&tail,--oldTail);
    (*t)=dq[oldTail];
    oldHead.u32=olatileRead(&head.u32);
    if(oldTail>((oldHead.u32)&0xFFFFFF))
      return true;
    volatileWrite(&tail,0);
    newHead.u32=0;
    newHead.u8[3]=oldHead.u8[3]+1; // ABA problem
    if(oldTail==((oldHead.u32)&0xFFFFFF))
      if(atomicCAS(&head.u32,oldHead.u32,newHead.u32)==oldHead.u32)
        return true;
    volatileWrite(&head.u32,newHead.u32);
    return false;}

  __device__ uint32_t popWarp(thread **tailPtr){
    if(dq==NULL||tailPtr==NULL)
      return 0;
    queueIdx oldHead,newHead;
    uint32_t oldTail,newTail;
    (*tailPtr)=NULL;
    oldTail=volatileRead(&tail);
    if(oldTail==0)
      return 0;
    oldHead.u32=volatileRead(&head.u32);
```

```
    // Determine the actual number of elements that can be acquired
    uint32_t popped=MIN(WARP_SIZE,oldTail-((oldHead.u32)&0xFFFFFF));
    newTail=oldTail-popped;
    volatileWrite(&tail,newTail);
    // The owner gets a pointer to the dequeue position from where elements can be taken
    (*tailPtr)=&dq[newTail];
    oldHead.u32=volatileRead(&head.u32);
    // Thieves always leave the last 2*WARP_SIZE elements to the owner
    if(newTail>=(((oldHead.u32)&0xFFFFFF)+WARP_SIZE))
      return popped;
    // Thieves already started stealing and do not know about concurrent pop operations
    if(newTail>((oldHead.u32)&0xFFFFFF)){
      newHead=oldHead;
      newHead.u8[3]++;
      if(atomicCAS(&head.u32,oldHead.u32,newHead.u32)==oldHead.u32)
        return popped;
      // The owner is the loser: recover the old tail-end index of the dequeue
      (*tailPtr)=NULL;
      volatileWrite(&tail,oldTail);
      return 0;}
    // Acquire the last element(s), and reset the dequeue
    (*tailPtr)=&dq[(oldHead.u32)&0xFFFFFF];
    volatileWrite(&tail,0);
    newHead.u32=0;
    newHead.u8[3]=oldHead.u8[3]+1;
    if(oldTail>=((oldHead.u32)&0xFFFFFF))
      if(atomicCAS(&head.u32,oldHead.u32,newHead.u32)==oldHead.u32)
        return oldTail-((oldHead.u32)&0xFFFFFF);
    volatileWrite(&head.u32,newHead.u32);
    (*tailPtr)=NULL;
    return 0;}}

 private:
  uint32_t size,tail;
  queueIdx head;
  thread *dq;
};
```

Listing A.2: Non-blocking dequeue implementation for Nvidia GPUs of compute capability at least 1.1.

On the CPU and the Xeon Phi we also extended the non-blocking dequeue implementation. It is used for the breadth-first search algorithm in Sec. 4.2.

A.3. Source Codes

Implementations of the data structures presented in this chapter (and in Chapter 3) can be found on the CD in `threadsafeDataStructures/[cpu,gpu,mic]` and `loadBalancing/[cpu, gpu,mic]`. On the top-level of each of the two directories is a `Makefile` that can be used for the execution of the test programs. Information on how to use these makefiles can be obtained with `make help`. Our makefiles do not handle all configuration setups. Switching between the setups using work stealing and the ones not using work stealing, for instance, must be adjusted in the source code. The source files contain many more `#defines` that can be set/unset or re-parametrized.

EXAMPLE: Enter directory `loadBalancing`. Type in

```
make run TARGET=gpu DEVICE=0 SCENARIO=sc1 THREADS=32
```

The test program for the load balancing scheme using scenario SC 1 + work stealing will start (see Sec. 3.5.1 and 3.5.2 for explanation). The test program is executed on GPU 0 (if present) with 32 threads (warps).

NOTE 1: The `Makefile` (most probably) needs to be adjusted to match the local installation of the libraries and SDKs.

NOTE 2: For all GPU benchmarks in Chapter 3 we have used Intel's `icpc 13.0.0` and Nvidia's `nvcc 4.1` (CUDA 4.1). For CPU and Xeon Phi benchmarks we have used Intel's `icpc 13.0.0`.

B. Ray Tracing

B.1. Work Distribution for Rendering the KingsTreasure Scene

Work distribution for rendering the KingsTreasure scene using setup 2 and 3, respectively. The description of the setups is given in Tab. 4.1.

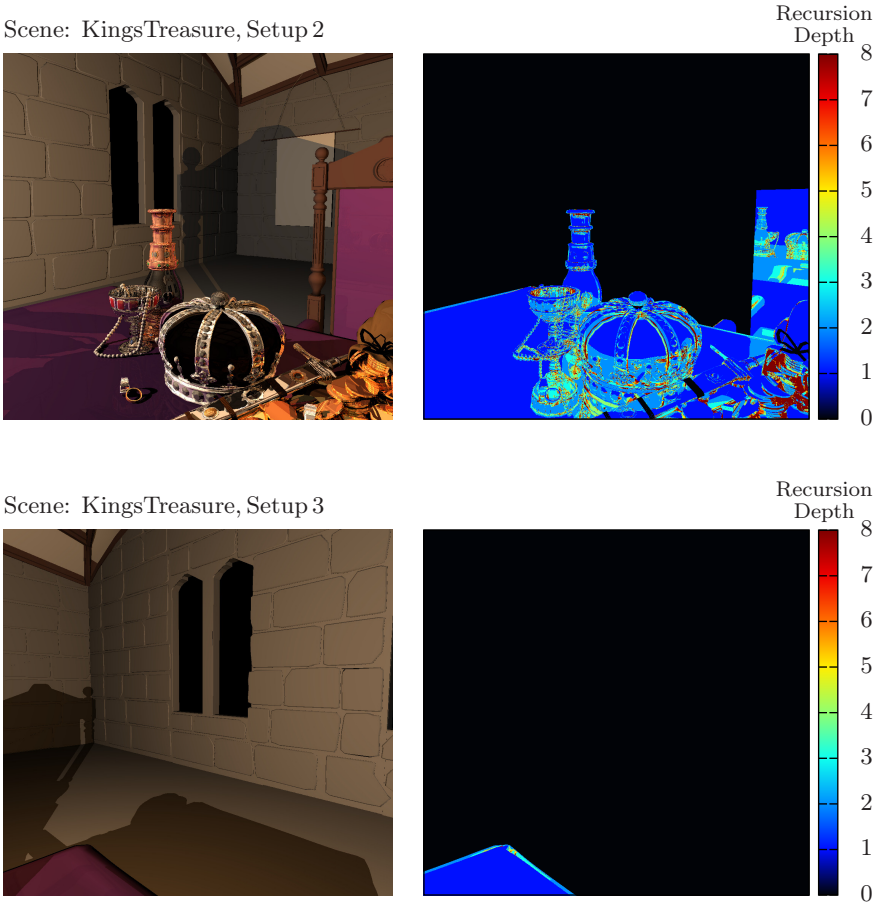

Figure B.1.: Work distribution for rendering the KingsTreasure scene using setup 2 and 3. Color values towards 'red' correspond to a lot of work, and values towards 'blue' correspond to little work.

B.2. Source Codes

All source codes used for benchmarking in Sec. 4.1.4 can be found on the CD in directory `rayTracing`. The `Makefile` placed in this directory can be used for building and executing the ray tracer. Type in `make help` for information on how to use the makefile. `make show` outputs the current configuration.

EXAMPLE: Enter directory `rayTracing`. Type in

```
make run TARGET=gpu DEVICE=0\
      ARGS="CONFIG=`pwd`/cfgFiles/setup_kingsTreasure_2.cfg"
```

The ray tracer should start rendering (on the GPU using device 0) the image defined in `setup_kingsTreasure_2.cfg`. We provide configuration setups for the KingsTreasure scene in subdirectory `rayTracing/cfgFiles`.

NOTE 1: We found the GNU compilers `g++4.4` to `g++4.7` produce somehow faulty executables of the ray tracing program when given the host code by Nvidia's `nvcc` compiler wrapper. We therefore provide a `Makefile` in `rayTracer/gpu/src` which compiles the entire code using Intel's `icpc` (a non-commercial version of this compiler is provided by Intel) except for the `*.cu` files. We did not have any trouble with the code generated in this way.

NOTE 2: The `Makefile`(s) (most probably) need(s) to be adjusted to match the local installation of the libraries and SDKs. Attend to not compile the ray tracer with the GNU compiler.

NOTE 3: For all GPU benchmarks in Sec. 4.1.4 we have used Intel's `icpc 13.0.0` and Nvidia's `nvcc 4.1` (CUDA 4.1)—for the ray tracer only the `*.cu` files were compiled with `nvcc`. For CPU and Xeon Phi benchmarks we have used Intel's `icpc 13.0.0`.

C. Breadth-First Search: Source Codes

Source codes implementing the BFS algorithm can be found in directory `bfs` on the CD. The `Makefile` placed in this directory can be used for building and executing our BFS programs. Type in `make help` for information on how to use the makefile.

EXAMPLE: Enter directory `bfs`. Type in

```
make run TARGET=xeonphi DEVICE=0\
      NX=100 NY=100 NZ=74 P=0.7 THREADS=120
```

The execution of the BFS algorithm should start on the Xeon Phi (device 0, if present) for a random graph with $100 \times 100 \times 74$ vertices, and edges between neighboring vertices established with probability $p = 0.7$ (see Sec. 4.2.2 for explanations). There are additional `#defines` in the codes that can be modified.

NOTE 1: The `Makefile` (most probably) needs to be adjusted to match the local installation of the libraries and SDKs.

NOTE 2: For all benchmarks in Sec. 4.2.2 we have used Intel's `icpc 13.0.0`.

Bibliography

[ABP98] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. pages 119–129, 1998.

[ATNW11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[BJK+95] R. D. Blumofe, Ch. F. Joerg, B. C. Kuszmaul, Ch. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient Multithreaded Runtime System. *SIGPLAN Not.*, 30(8):207–216, August 1995.

[BL93] R. D. Blumofe and Ch. E. Leiserson. Space-efficient Scheduling of Multithreaded Computations. pages 362–371, 1993.

[BL99] R. D. Blumofe and Ch. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. *J. ACM*, 46(5):720–748, 1999.

[BNP12] M. Burtscher, R. Nasre, and K. Pingali. A Quantitative Study of Irregular Programs on GPUs. pages 141–151, 2012.

[Bre74] R. P. Brent. The Parallel Evaluation of General Arithmetic Expressions. *J. ACM*, 21(2):201–206, 1974.

[CGSS11] S. Chatterjee, M. Grossman, A. Sbirlea, and V. Sarkar. Dynamic Task Parallelism with a GPU Work-Stealing Runtime System. 2011.

[Chr12] G. Chrysos. Intel Xeon Phi Coprocessor (Codename Knights Corner). 2012.

[CRL90] Th. H. Cormen, R. L. Rivest, and Ch. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 1st edition, 1990.

[CT08] D. Cederman and P. Tsigas. On Dynamic Load Balancing on Graphics Processors. pages 57–64, 2008.

[DLS+09] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha. Scalable Work Stealing. pages 53:1–53:11, 2009.

[dRu12] 3d Renderer. `http://www.3drender.com/challenges/index.htm`, 2012.

[GLFR12] T. Gautier, F. Lementec, V. Faucher, and B. Raffin. X-Kaapi: A Multi Paradigm Runtime for Multicore Architectures. (RR-8058):16, February 2012.

[HKB12] A. Heinecke, M. Klemm, and H. J. Bungartz. From GPGPU to Many-Core: Nvidia Fermi and Intel Many Integrated Core Architecture. *Computing in Science and Engineering*, 14:78–83, 2012.

[Int12] Intel. Intel Xeon Phi Coprocessor 5110P, Prodct Brief, 2012.

[LWH10] L. Luo, M. Wong, and W. Hwu. An Effective GPU Implementation of Breadth-First Search. pages 52–55, 2010.

[Nvi09] Nvidia. Fermi Compute Architecture Whitepaper, v1.1. October 2009.

[Nvi12] Nvidia. Nvidia CUDA C Programming Guide, v5.0. July 2012.

[SAG+05] P. Shirley, M. Ashikhmin, M. Gleicher, S. Marschner, E. Reinhard, K. Sung, W. Thompson, and P. Willemsen. *Fundamentals of Computer Graphics, Second Ed.* A. K. Peters, Ltd., 2005.

[SHT+12] R. Solcà, A. Haidar, S. Tomov, J. Dongarra, and T. Schulthess. A Novel Hybrid CPU-GPU Generalized Eigensolver for Electronic Structure Calculations Based on Fine Grained Memory Aware Tasks. *Supercomputing '12 (poster)*, November 2012.

[SM03] P. Shirley and R. K. Morley. *Realistic Ray Tracing.* A. K. Peters, Ltd., 2003.

[SRM13] T. Schütt, A. Reinefeld, and R. Maier. MR-Search: Massively Parallel Heuristic Search. *Concurr. Comput.: Pract. Exper.*, 25(1):40–54, January 2013.

[SW87] R. H. Swendsen and J. Wang. Non-Universal Critical Dynamics in Monte Carlo Simulations. *Phys. Rev. Lett.*, 58:86–88, Jan 1987.

[SYD09] F. Song, A. YarKhan, and J. Dongarra. Dynamic Task Scheduling for Linear Algebra Algorithms on Distributed-Memory Multicore Systems. pages 19:1–19:11, 2009.

[TLO12] S. Tzeng, B. Lloyd, and J. D. Owens. A GPU Task-Parallel Model with Dependency Resolution. *Computer*, 45(8):34–41, 2012.

[TZ00] P. Tsigas and Y. Zhang. Evaluating the Performance of Non-Blocking Synchronisation on Modern Shared-Memory Multiprocessors. 2000.

[WF12] F. Wende and T. M. Feist. Ray Tracing on GPUs. `http://www.inf.fu-berlin.de/lehre/SS12/SP-Par/download/FeistWende.pdf`. 2012.

List of Figures

List of Tables

Acknowledgment

I thank Prof. Dr. Helmut Alt (FU-Berlin) and Prof. Dr. Alexander Reinefeld (HU-Berlin, Konrad-Zuse-Zentrum für Informationstechnik Berlin (ZIB)) for enabling me to work on this topic, for having been there for questions, and for having given valuable suggestions to this thesis.

Special thanks go to Dr. Thomas Steinke (ZIB) who was intensively involved in the supervision of this thesis. I also want to give thanks to him for interesting discussions and for proofreading of many parts of this document.

Further thanks go to Mr. Igor Merkulow for proofreading and for some interesting ideas that had some influence on this work.

I also give thanks to Dr. Ludmila Scharf and Mr. Paul Seiferth for proofreading and discussion.

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Ort, Datum

Unterschrift