FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's thesis

Sune Tougaard-Andersen

Efficient Chip Multi Processor Programming
Programming a Multi-Core Processor

Supervisor: Jyrki Katajainen

Date of submission: September 2011

Abstract

In this work a realistic machine model, the CMP-model, is investigated. This model captures the cache hierarchies on mainstream CMPs as well as the ways that these caches interacts.

A parallel programming library for benchmarking is presented. The presented library introduces policy based scheduling allowing fair evaluation of scheduling algorithms. The parallel-depth-first scheduler theoretically perform better than the widely used work-stealing scheduler, in the CMP-cache model. Both schedulers are implemented in the benchmarking library.

Efficient parallelizations of the well-known sequential sorting algorithms quicksort and multi-way mergesort are analysed based on the CMP-cache model. The parallel quicksort algorithm is based on a parallelization of the in-place sequential partitioning algorithm. The parallel multi-way mergesort is based on a f-way partitioning algorithm.

The presented library is used to evaluate the two analysed parallel sorting algorithms using both the implemented schedulers. We find that efficient parallel sorting algorithms are limited by memory access. We also find that the algorithmic techniques know from the external-memory model can be used to gain some performance on a CMP.

Resumé

Den mere realistiske model, CMP-cache modellen bliver analyseret. CMP-cache modellen modellerer både det cache-heraki, som moderne CMP'er har, og de interaktioner dette medfører.

Et bibliotek til evaluering af parallelle algoritmer og skeduleringsalgoritmer præsenteres. Politikbaseret skedulering introduceres. Politikbaseret skedulering gør det muligt at evaluere skeduleringsalgoritmer refærdigt. Skeduleringsalgoritmen, parallel-dybde-først, er teoretisk bedre i CMP-cache modellen end den udbredte work-stealing algoritme. Begge algoritmer er implementeret i det præsenterede evalueringsbibliotek.

Effektive paralleliseringer af de to velkendte sorteringsalgoritmer, quicksort og flervejs-mergesort, analyseres i forhold til CMP-cache modellen. Den parallelle quicksort algoritme er baseret på en parallel version af den sekventielle in-place partitioneringsalgoritme. Flervejs-mergesort algoritmen er baseret på en f-vejs partitioneringsalgoritme.

Det præsenterede bibliotek bruges til at evaluere de to analyserede sorteringsalgoritmer for begge af de implementerede skeduleringsalgoritmer. Vi finder, at de effektive parallelle sorteringsalgoritmer er begrænsede af hukommelsens hastighed-ed. Vi finder ydermere, at teknikkerne, fra den eksterne hukommelses-model, kan bruges i forbindelse med CMP programmering.

Acknowledgements

I would like to thank my supervisor Jyrki Katajainen for his encouragement and guidance during the writing of this thesis. I would also like to thank Henrik Tougaard for the conversions that helped me structure mt thoughts and for proof reading the drafts. Lastly, but not least, I would like to thank my fiancee Maria for her patience and forbearance whilst I lost myself in the making of this thesis.

Contents

1 Introduction

The current developments in hardware is towards processors with more cores, commonly know as *multi-core processors*. A multi-core processor is a single chip with several physical processors, they are for this reason also know as *chip multi processors* (CMP). These processors require special programming techniques to obtain the full performance potential of the processor. Parallelism is an old algorithmic problem that have become even more relevant. Parallelism have previously been associated with cluster computing, and many of the techniques have therefore been developed for this setting. As the number of core contained in a ordinary CPU increases, a new setting of parallel programing emerges. While the techniques for cluster computing can be used successfully on a CMP the question of whether other programing techniques would result in better performance arises.

As multi-core processors have become the common have clusters now become based on CMPs. This makes it more relevant to investigate techniques for efficient programming of multi-core processors, as any possible increases in performance for a single CMP might also result in increases of performance for CMP based clusters.

The use of cache hierarchies have become a necessary due to the latencies on memory accesses. Computational models have been developed to identify how these cache hierarchies are best utilized. This thesis is an investigation in how the current multi-core processors are best programmed.

1.1 Contributions

This thesis present a library for benchmarking of scheduling algorithms. This library use a parametrisation of scheduling algorithms that allow the scheduler to be chosen as a policy. This enables fair comparisons of scheduling algorithms as exact same program can be executed with different schedulers.

Two efficient parallelizations of well-known sequential sorting algorithms are analysed in a multi-core processor setting. The sorting algorithms are benchmarked using two different scheduling algorithms.

2 Models of Computation

This Chapter introduce two memory models and the programming techniques they lead to are discussed. These introduce the notation and concepts used in the rest of the thesis. The Chip-Multi-Processor-Cache model is then introduce and compared with the reality of actual hardware.

The external-memory model Section is written from [1], [19], [11], [24] and [12]. The Ideal-cache model Section is written from [13] [15] and [13]. The CMP-cache model Section is written from [2], [9] and [13]. The reality Section id written from [2], [9], [13], [8] and [18].

2.1 External-Memory Model

The benefits of the widely used RAM model is its simplicity. One of the reasons for simplicity is the assumption of unlimited memory. In reality this is not true; computers have a limited main memory and a large secondary storage, usually in the form of disks. When data is too large to fit in memory, the RAM model can no longer be used for efficient programming.

A disk stores data on a number of rotating *platters*. Data is written to and read from the platters by read/write heads located on a movable arm. Data on the platters is stored in circles called *tracks*. Tracks are located on each platter such that when the arm is stationary the whole track will pass under the read/write head. Disks are cheaper and have a higher capacity than main memory, but they have a slower access time. The access time is limited by the rotation speed of the platters and the speed of the arm. In order to amortize the large access time, data is transferred to and from the disk in blocks of contiguous memory segment, called *pages*. Transferring pages instead of smaller data segments amortizes the access time if the data exhibits good locality. The assumption is that the data used by a program is placed on the disk near to previously used data.

Transferring a page to memory from disk or to disk from memory is called an I/O operation, or just an I/O. Because I/O operations are a performance bottleneck, we are interested in a way to argue about the number of page transfers that are performed by a program. We therefore modify the RAM model to include the limited size of main memory and the need for I/O operations. The new model is called the external-memory model and includes the following parameters, all of which are measured in units of data items [1]:

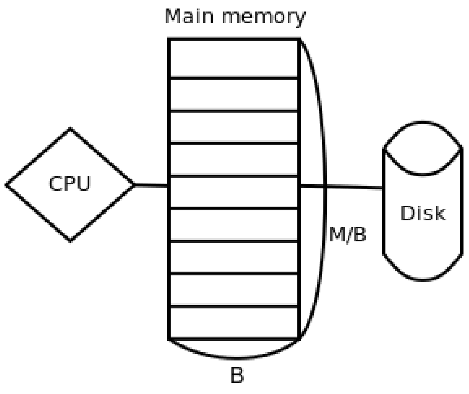


Figure 2.1: Diagram of the external-memory model.

N the problem size.

M the size of main memory.

B the size of a page.

Besides the three parameters above, m and n are also used in the literature, as meaning the number of pages that can fit in memory, M/B, and the number of pages that the problem size fills, N/B, respectively. It is assumed that $M < N$, $M > B$ and that the size of external memory is infinite. When a program accesses data not stored in main memory, the page containing the is data read from disk and loaded into main memory. The access to page not stored in memory is called a *page fault*. Page faults are closely related to I/Os. The page fault is the access to a memory address not stored in main memory. Whereas an I/O is the act of transferring the requested page from disk to memory. Each page fault thus causes an I/O.

A common algorithm template in the external-memory model is a scan. In a scan each data item is processed once. Data items are processed in an order, such that each data item in a page is processed before a new page is accessed. We can treat the page as a buffer for a single data item. The program accesses the data item one at a time, when the buffer is empty a new page is loaded into memory. This pattern performs a linear number of I/Os which is optimal while still being simple.

2.1.1 B-Tree

A B-tree is a balanced tree with all leaves on the same height in the tree. The fanout of the tree is designed such that a node can fit in a single page, i.e. $O(B)$. A node consists of an ordered set c of pointers to the children nodes and an ordered set r of records. The records are used to separate the children nodes. Every record in the subtree rooted by the ith child is smaller than the corresponding record entry. When searching the tree this allows the child the search should proceed in, to be found by binary search on the set of records. A node is mapped to one continues memory section located in a single page.

Searching a B-tree takes $\mathcal{O}(\log_B(N))$ I/O operations. The height of the tree is $\mathcal{O}(\log_B(N))$ as the fanout is $\mathcal{O}(B)$ and each node access requires at most one I/O operation. In practice B-trees are very fast and thus widely used in both databases and file systems. The height of a B-tree is usually very limited.

2.1.2 Multi-Way Mergesort

Mergesort is a well-known sorting algorithm based on the divide-and-conquer paradigm. The algorithm works by dividing the input into sections that are trivial to sort, and then merge sorted sections into a single sorted section remains. This algorithm is time optimal in the RAM model where binary merge is normally used repeatedly. In the external-memory model binary merging leads to a non-optimal number I/O operations. The problem with the binary merge is that each data section is used in a merge many times and is therefore moved to and from disk many times. To obtain better I/O performance each page of data should be use as few times as possible. This can be achieved by using a multi-way merge instead of a binary merge procedure.

The multi-way merge procedure works by scanning over all the sorted sections. The merging procedure keeps a buffer page for each section and a buffer page for the merged output. This makes merging possible while only needing one page of data from each section being merged. When all data values in a buffer page have been merged a new page of data, from the same section, is loaded into memory. The output buffer page is managed likewise, when it is full the page is written to disk and a empty page becomes the new buffer. Because each section can be scanned using only one page, it is possible to merge as many sections as the number of pages that can fit into memory.

The actual merging is done using a priority queue. The smallest element from each sorted section is inserted into the queue. The minimum is then removed from the queue and inserted into the output buffer. The next element in the section, from which the previously removed element belonged, is then inserted into the queue.

We can now make a mergesort algorithm for the external-memory model, using m-way merging. This is done in a bottom-up fashion. First we divide the input into sections of $\mathcal{O}(M)$ elements that are sorted with a normal sorting algorithm. We then recursively merge the sorted sections with m-way merging until only one section remains. Each merging phase reads and writes all elements and thus each merge phase uses $\mathcal{O}\left(\frac{2N}{B}\right)$ I/O operations. The number of merging phases is $\mathcal{O}\left(\log_m\left(\frac{N}{M}\right)\right)$ and thus the algorithm uses $\mathcal{O}\left(\frac{2N}{B}\log_m\left(\frac{N}{M}\right)\right)$ I/O operations and $\mathcal{O}(n\log n)$ comparisons. Notice that very few merge phases are needed in practice, if $N \leq M^2/B$ then only one merging phase is needed.

2.2 Ideal-Cache Model

Algorithms in the external-memory model directly use the parameters M and B. For this reason these algorithms are called cache aware, as the algorithms are aware of the parameters of the cache they use. The implementation of an algorithm must therefore determine these parameters, before the algorithm can run optimally. These parameters must be redetermined each time the hardware or underlying systems are changed. We would like to be able to construct algorithms that are cache optimal without determining the cache parameters. This kind of algorithms are called cache oblivious, as they do not directly use the cache parameters.

Today's computers contain several caches used to conceal latency. As described in Section 2.1, disk-based memory is slow, but main memory is also too slow to be used directly for computations. To hide latency a hierarchy of caches is used. Each level of cache closer to the processor has lower capacity and lower access time. Because cache-oblivious algorithms are cache optimal for all cache parameters, they are optimal on all levels in the cache hierarchy. This holds under the assumption that all cache levels have the *inclusion property.* which requires that the smaller cache closer to the processor contains a subset of the data in the larger cache.

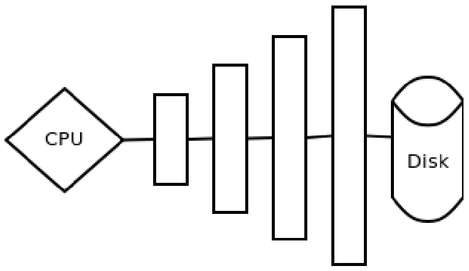


Figure 2.2: Diagram of the ideal-cache model.

In the ideal-cache model we are still interested in the number of I/Os as a performance metric. In order to be able to analyse the I/O performance of cache-oblivious algorithms, the same model parameters as in the external-memory model are used. The meaning of the model parameters has changed slightly to be more natural [12].

M the size of the cache

B the size of the cache lines

N problem size

We still assume that data is moved to and from the cache in blocks; these blocks, are called cache lines. To simplify the model the cache is assumed to be ideal. An ideal cache obeys following two properties:

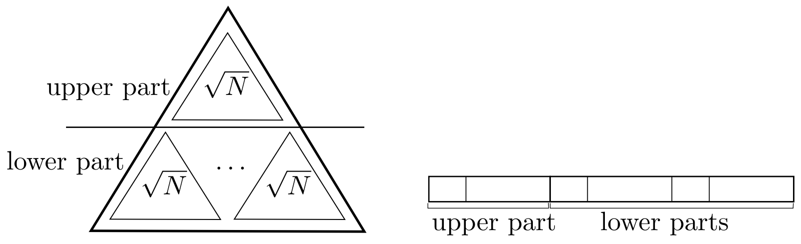
Optimal replacement: When the cache is full and a cache line needs to be loaded, another cache line must be evicted. The strategy for selecting which cache line to evicted is called the *replacement strategy.* An ideal cache is assumed to use optimal replacement strategy. It is assumed that the replacement algorithm can see the future and selects the cache line used furthest in the future for eviction. This is not possible in practice, where an online algorithm must be used instead. Among the practical algorithms are the least recently used which selects the block that has been used the furthest in the past, and first in first out.

Fully associativity: It is assumed that all cache lines can be placed anywhere in the cache. This assumption is not true in practice where the cache has a limited associativity to improve performance.

Even though the ideal-cache assumptions are not true in practice, they allow us to construct algorithms that are good in practice. It can be shown that they only decrease the performance by a constant factor, assuming that the number of cache misses have a polynomial dependency on the cache size [15]. One further assumption is often used in the analysis of cache oblivious algorithms the *tall cache assumption.* This assumption says that the cache can contain more pages than the size of a page, $M > \Omega(B^2)$. In practice this assumption is usually true.

2.2.1 van-Emde-Boas Layout

In the external-memory model B-trees made it possible to search using $\mathcal{O}(\log_B N)$ I/O operations. Searching with the same I/O bound is possible in the ideal-cache model. B-trees used the page size to maximise the branching factor, as we do not know the page size, this is not possible in the ideal-cache model. We are therefore forced to go back to a binary search tree. To achieve better I/O performance for searching, we optimise the memory layout of the search tree. We map the tree to a single one dimensional memory section, like an array. The better I/O performance can then be achieved by packing the memory sections optimally. This mapping from a tree to a memory section is known as the van-Emde-Boas layout, shown on Fig 2.3. The van-Emde-Boas layout uses the same idea as van-Emde-Boas trees, which are divided horizontally in the middle into an upper and a lower part. The upper section is put first in the memory section then all the lower parts. Each part is then recursively mapped in the same way. The upper part contains $\sqrt{N}$ elements and the lower part consists of $\sqrt{N}$ subtrees each containing $\sqrt{N}$ elements.



(a) Tree division. (b) Memory mapping to contiguous memory.

Figure 2.3: The recursive division of a tree and the corresponding mapping to memory in the van-Emde-Boas layout.

Searching in a tree with van-Emde-Boas layout takes $\mathcal{O}(\log_B N)$ I/O operations. We argue this using subtrees containing at most B elements with height $\mathcal{O}(\lg B)$. A subtree is laid out in memory in a continuous section. Because the size of the subtree is B elements, it can fit into a single cache line. A subtree can reside in at most $\mathcal{O}(1)$ cache lines, the alignment of the subtree might lead this to be two instead of single cache line. The search path from the root to a leaf contains $\mathcal{O}(\lg N)$ nodes and will thus pass through $\mathcal{O}(\log_B N)$ subtrees. Accessing each subtree results in the transfers of at most two cache lines.

2.3 Chip-Multi-Processor-Cache Model

Most computers have more than one processor today, usually in the form of a *chip multiprocessor* (CMP). A CMP is a single chip that contains more than one processor, also called a *core*. A CMP is thus able to execute a number of instructions concurrently while avoiding having more than one processor chip.

We would like to expand the previously discussed memory models to include the possible concurrency that CMPs offer. Much work has gone into this goal resulting in several different models. These models differ on some key points and thus obtain different results in some key areas. One of the key differences of the models are whether the memory is shared or private. A *shared cache* lets all processors access and modify it. A *private cache* gives each processor its own cache that only that processor can use.

The CMP-cache model is a combination of both cache types, which leads to a more realistic model. Figure 2.4 show a diagram of the CMP-cache model. The CMP-cache model has P processors and a three-level memory hierarchy, unlike the ideal-cache model, the cache line sizes are assumed to be the same for all cache levels and smaller than the page size. The cache closest to the processors is private to each processor and is referred to as the *L1 cache*. All P L1 caches have the same size of C_1 cache lines. The next cache in the hierarchy is called the *L2 cache*. It is shared between all P processors. The size of the L2 cache, C_2, is assumed to be larger then the combined size of the L1 caches $C_2 \geq P \cdot C_1$. It is also assumed that the problem is too large to fit in L2 cache, so $N > C_2$.

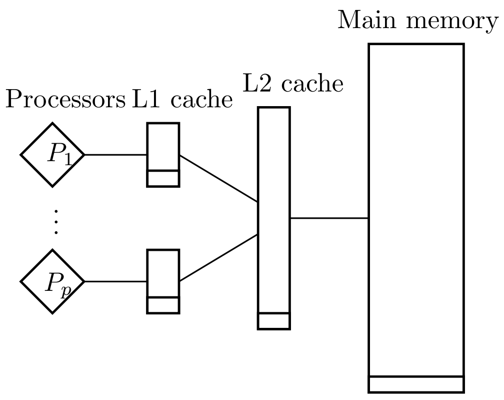


Figure 2.4: Diagram showing the CMP-cache model. The block size is shown by the square in the bottom of each cache.

Before any data is used by a processor the cache line, in which the data value resides, is placed in the L1 cache. A cache line can be present in more than one L1 cache as long as it only is read from. If a processor wants to write to a cache line, the cache line must first be evicted from all other L1 caches , this process is descried in more detail in Section 2.4.

The CMP-cache model defines two cache complexity metrics; the L1 cache complexity and the L2 cache complexity. These are defined in terms of L1 and L2 cache misses respectively. When a processor requests read or write access to a cache line, the cache line can either be in L1 cache or not. If the cache line is not cached, it must first be loaded into cache; this is called a *cache miss*. If the cache line is in a cache, a *cache hit* is said to happen. An L2-cache miss occurs when a requested cache line is neither in any L1 cache or in the L2 cache. Note that an L2-cache miss does not occur if the cache line is present in a L1 cache belonging to another processor. The definition of L2-cache misses reflects the fact that most CMPs allow a processor to fetch cache lines from the private cache of another processor faster than fetching it from main memory. The two cache complexity metrics reflect the limited bandwidth to off-chip memory.

2.3.1 Types of Cache Sharing

There are two principal way processors on a CMP can share a shared cache [9]:

Cooperative sharing is when the processors use the same data.

Competitive sharing is when the shared cache contain on shared data. In competitive sharing each processor treats the shared cache as a private cache. The processors thus competes for space in the shared cache.

Competitive sharing can lead to increased contention on the off-chip bandwidth as all processors tries to fetch new data into cache but the off-chip bandwidth is limited and can only be accessed by a limited number of concurrent accesses. Because the cache is shared can competitive sharing lead to cachelines being evicted prematurely because another processor have fetched a more data into cache. competitive share thus leads to a poorer performance than cooperative sharing.

Cooperative sharing decrease the contention on the off-chip bandwidth as the memory is accessed in a sequential scan. Cooperative sharing make it possible for P processors to effectively a smaller cache than is possible with competitive sharing.

Cooperative sharing is preferable over competitive sharing, but to be able to solve a problem in parallel the problem must be divided into independent parts that can b e preformed concurrently. This division of a problem means that it can be hard to actually solve the problem in a way that both allow parallelism and cooperative cache sharing.

2.4 Reality

CMPs include a cache hierarchy to hide the latency of memory access. The caches in the hierarchy on a CMP are named after their proximity to a processor. The closest cache is the L1 cache, the second closest cache is called L2 and so on. This naming scheme is used independently on whether a cache is shared or private, as the cache hierarchy is usually equivalent for all processors on a chip. The actual cache layout and sizes vary greatly. To give an idea the Intel Nethalem micro-architecture is used as an example. Generally the L1 caches are small. Nethalem has a L1 cache of 32KB for data and 32KB for instructions. The L2 cache is larger and can either be private to each core or shared between a subset or all cores. Nethalem has a 256KB private L2 cache for both instructions and data. When a L3 cache is used, it is larger than the L2 cache. Nethalem uses a 8MB shared L3 cache.

Data is stored in a cache in blocks called *cache lines.* Unlike main memory, cache is not fully associative but is set associative. Set-associative cache maps a memory address to a set of cache lines where the data is stored. This cache design increases the performance of the cache [13, 15], but can lead to premature eviction of a cacheline because the two cachelines map to the same position in cache and thus cannot be in cache simultaneous.

In order to make cache usage transparent for the programmer, synchronization between caches is needed. This effect is known as *cache coherence* and is especially important for CMPs. The *MESI* cache-coherence protocol has been developed to manage the cache coherence process. It is named after the four states a cache line can be in: modified, exclusive, shared, and invalid. A cache line is *modified* if a processor has written to it, a modified cache line is also exclusive. A cache line is *exclusive* when it is only in the private cache of one processor. A cache line is *shared* when it is in more than one processors private cache. To modify a shared cache line the processor must first take ownership of the cache line by changing the state of the cache line to exclusive. To *take ownership* of a cache line the processor sends a request for ownership (ROF) message to the other processors that then mark their own copy of the cache line as invalid. An *invalid* cache line is known to be wrong and the processor must therefore fetch the new version of the cache line before accessing it.

Writing to a shared or invalid cache line is thus more expensive and as synchronisation is necessary restricts the possible concurrency. If two processors attempt to write to the same cache line, one processor will take ownership of it, making the other processor mark its own copy as invalid. The first processor can then modify the cache line. The second processor can now take ownership of the cache line and then modify it. This process can remove otherwise available concurrency and can even occur if two unrelated variables share a cache line.

Because a processor must take ownership over a cacheline before any modification can take place, can cooperative cache sharing result in a performance degradation if all processors attempts to modify the same cacheline. This can take place even if all processors tries to modify different elements that resides in the same cacheline.

At hardware level main memory access is controlled by a memory controller. Access to the memory controller is traditionally done through the front-side bus. On computers that use one or more CMPs, this design leads to contention on the memory access. To mitigate the contention most modern architectures utilize two techniques. The first is dual or triple channel memory. *Dual-channel memory* was originally introduced to increase the bandwidth for sequential access to the main memory, but today it is mostly used to allow for concurrent accesses.

Dual- and *triple-channel memory* partition the main memory into two or three sections respectively. Each section has its own memory controller, allowing concurrent access to the different memory sections and thus lowering contention. The second technique for lowering the contention of main memory access is *non-unified memory architecture* (NUMA). In NUMA each processor chip has its own main memory and memory controllers. This leads to less contention on the memory access, but higher latency when a process accesses a memory address that is stored in the memory of another processor. NUMA makes it more important that processes that share memory sections are scheduled on the same chip.

3 Load Balancing

It is not always possible to construct a parallel algorithm such that all processors have n equal amount of work, and the work load need to be balanced to avoid ideal processors. Load balancing is performed by algorithms that define what each processor should do. The act of assigning work to each processor is called *scheduling*. In this Chapter we introduce scheduling. The scheduling theory is base on a model of parallel programs. This model is used through out the thesis, and is introduced here as it is very related to the understanding of scheduling.

The widely used scheduler, work-stealing, is introduced in this chapter. Work-stealing is a efficient scheduler that minimizes the amount of communication between processors. Work-stealing is also used when only a single CMP is available, but this is not necessarily the best choice. The parallel-depth-first scheduler achieves better cache bounds than work-stealing under the CMP-cache model.

The Directed-Acyclic Graph Model and scheduling theory sections are based on [5], [4], [2] and [3]. The work-stealing Section is based on [5] and the parallel-depth-first scheduler Section is based on the [4], [2] and [3].

3.1 Directed-Acyclic Graph Model

Concurrent programs are often modelled as a directed-acyclic graph (DAG), defined as a set of vertices V and a set of directed edges E. Each vertex represents a unit-time *action* and each directed edge a *dependency*. For the two vertices u and v, where $(u, v) \in E$, the action u must be completed before v can be started. The DAG model gives a natural and intuitive understanding of parallel programs.

From the DAG model two useful properties for a parallel program can be defined:

Work is the total number of vertices in the graph or the number of unit time actions that the program contains as a whole and is, for a DAG G, denoted $work(G)$. The $work(G)$ of a DAG is thus the time a program will use when only a single processor is available.

Depth is the longest path in the graph and is, for DAG G, denoted $depth(G)$. The depth of a program is thus the time the program will use if an infinite amount of processors would be available.

An action with more than one child, is said to *Spawn* the Children actions. Spawning models possible parallelism, as all children of an action can be started concurrently if no other dependencies prevent this.

The DAG model does not limit the number of parents an action can have or in other way limit the structure of the graph. The requirement that the graph is acyclic is necessary as a cycle would make scheduling the graph impossible. The edges represent all types dependencies but can be classified by the dependency type the edge represents. There are three types of data dependencies:

Data dependency: An action requires that some data is made by another action. If data dependencies are not restricted, they can make the graph impossible to schedule efficiently.

Sequential dependency: Models the sequential structure of actions in a program, ie. the ordering of actions expressed by a sequential function. An action can logically only have one outgoing and on ingoing sequential dependency.

Spawn dependency: An action that spawns another action, must be completed before the spawned child can be started. Spawn dependencies are similar to Sequential dependencies, in that an action cannot be started before it have been spawned. Unlike sequential dependencies are an action allowed to have multiple outgoing spawn dependencies.

When all dependencies for an action is fulfilled and the action can be started, the action is said to be *ready*.

The DAG model is not restricted to offline scheduling, where the whole DAG is known before computation starts. But it also works well in the online setting, where the DAG is revealed as computation progresses.

When working with concurrent programs it is more natural to divide the program into parts that are bigger than unit time actions. This is done by logically dividing the DAG into segments of actions that is ordered by sequential dependencies. This segments are small sequential program segments or functions, that can be though of as super actions. Working with this larger segments makes writing programs more natural, as the program can be expressed in sequential functions and their dependencies. There are two ways to structure these super actions:

Threads: The DAG is thought of as consisting of sequential program threads. When a thread spawns, the threads splits into two the child thread and the continuation of the parent thread. A thread is completed when the last action in the thread is completed. A parent thread can only complete when all its children have completed. Figure 3.1 show the threads in a DAG.

Task: The DAG is thought of as consisting of small sequential tasks. When a task spawns a set of children tasks the children task are thought as part of the parent task. The spawn action pauses execution of the parent task, when all children tasks have completed the execution of the parent task is continued. Figure 3.2 show the division of a DAG into tasks.

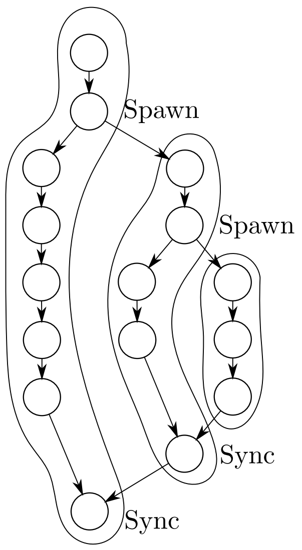


Figure 3.1: Visualisation of the spawning and synchronising of threads that the work stealing scheduler use. Each thread is encircled. Notice that the parent thread continues when a child thread is spawned. The completion of a child thread is marked with 'Sync', as described in Section 3.2.1

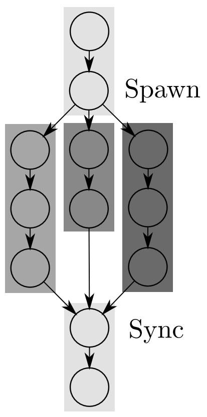


Figure 3.2: Visualisation of the spawning and synchronising of tasks. Each circle is an action and each grey box marks the tasks. The different tasks are all coloured in different shades. Notice that execution of the parent task is suspended after spawning the child tasks and is resumed after all child tasks have died.

3.1.1 Performance Metrics for Parallel Programs

The point of parallel computing is to obtain faster execution of a program. The increase in execution speed due to an increase in the number of used processors is called *speedup*. The running time of an algorithm on a number of processors is written as $T_p(n)$, where p is the number of processors used and n is the input size of the problem instance. The running time for the best known sequential algorithm, that solves the same problem, is denoted $T_* (n)$. The speedup of an algorithm is defined as:

$$Speedup_p (n) = \frac{T_* (n)}{T_p (n)} = \frac{work(n)}{depthn}$$

As the parallel and the sequential algorithms must solve the same problem they must perform the same computation. The theoretical optimal speedup is therefore linear in the number of processors used and is often referred to as *linear speedup*. It is possible to obtain *super linear speedup* in practice either due to caching effects or when the solution to a problem can be found with out all computations needed to be done e.g. searching where each processor can search a subsection of the input, it is the possible for one of the processors to find the solution, before a sequential algorithm would, and notify the other processors resulting a much faster run-time in some cases.

Because an optimal sequential algorithm define the minimum needed computational work needed to bee done to solve a problem, the parallel algorithm is said to be *work optimal* if it preform the same amount of work as the sequential algorithm.

An algorithm or part of an algorithm that cannot be parallelized is known as inherently sequential. Ahmdal made an important observation in 1967 about the effect of an inherently sequential part of an algorithm on the speedup of the algorithm as an whole. This observation is known as Amdahl's law and states that if a fraction f of a program can be parallelized with a speedup of $Speedup^f$ the speedup of the program as a whole will increase by:

$$Speedup_\infty (n) = \frac{1}{(1-f) + \frac{f}{Speedup^f}}$$

This function decreases very fast when decreasing f. Amdahl's law and the ideal speedup defines a maximal speedup for a program using P processors.

$$Speedup_p(n) = \frac{1}{(1-f) + \frac{f}{P}}$$

When P goes to infinity the speedup goes to $1 - f$, the fraction of the program that is inherently sequential. Amdahl's law shows that it is important to parallelize as much of a program as possible. Any sequential part of a program will severely limit both the obtainable speedup and the practically usable number of processors. An example of how limiting a sequential fraction of an algorithm is given in Section 4.1.

The speedup of a parallel algorithm does not increase beyond a certain number of processors, using more processors after this limit do not improve performance and can in some cases degrade performance. The parallel algorithm is said to *scale* to this limit. It is number of processors an algorithm scales to express how much available parallelism the algorithm have. Scaling limit is important for the performance of an algorithm, if the number of usable processors is very low then it does not matter if the algorithm achieves good speedup.

3.2 Scheduling Algorithms

Often the amount of available parallelism is much greater than the number of processors. A way of balancing the work is then needed to insure that all processors are running for as long as possible. An example where work balancing is needed can be seen in the parallel quicksort algorithm described in Section 4.2.2. This work balancing is achieved by scheduling which task of the program DAG should run on which processors. The topic is thus known as scheduling and the algorithms for making the schedules are known as scheduling algorithms.

A *schedule* of a DAG consists of a sequence of steps $1 \dots \tau$. Each step defines a set of vertices, $V_i, i = 1 \dots \tau$, that are scheduled at that step. A schedule must obey the following two properties:

1. Any vertex is scheduled exactly once, i.e. an action is a member of exactly one set of actions scheduled by in a step. The sets V_i thus partition the DAG into disjoint pieces
2. An action can only be scheduled in a step, if all the ancestors of that action are scheduled at an earlier step, i.e. if, for a DAG with a set E of edges, $(u, v) \in E$ and u is scheduled at step V_i then $v \in V_j, j < i$

The *completed set* $C_i = V_1 \cup V_2 \cup \dots \cup V_i$ for a schedule $\mathcal{S}$ on a DAG G is the set of actions completed at step i. The schedule $\mathcal{S}$ thus induces a partial order of the actions in G and a sequential schedule induces a complete order of the actions in G. An action u *precedes* an action v if $u \in V_i$ and $v \in C_{i-1}$.

Scheduling introduces overheads on the computation. To mitigate these overheads larger tasks or threads may be scheduled instead of the unit-time actions.

3.2.1 Work-stealing

The work-stealing scheduling algorithm schedules threads, instead of actions. A thread is said to be *alive* while a processor is executing the thread. A thread *stalls* if the next action in the thread have an unresolved dependency and the next action in the thread is not ready, preventing the execution of the thread to continue. When the dependency is resolved the thread becomes ready.

The work-stealing scheduler makes some simplifying assumptions on the structure of the DAG that is scheduled. These assumptions must be fulfilled by the DAG of the program that is scheduled.

- A thread is only allowed to spawn one thread at a time. Spawning multiple threads can be accomplished by repeatedly spawning.
- When a thread is spawned both child thread and the parent thread must be ready. The parent thus cannot sync with the spawned child thread immediately after the child is spawned.
- An action can only have a constant number of synchronisation edges incident to it.

- A synchronisation edge between two threads is only allowed from a thread to the thread parent.

The work-stealing scheduler uses a distributed approach for scheduling; each processor has a schedule of its own. Whenever the schedule of a processor become empty, the processor attempts to steal work from a victim processor.

Each processor uses a deque to store the ready threads. The front of the deque is used by other processors and the back of the deque is used by the owning processor. Whenever a thread spawns a child thread, the parent thread is pushed onto the back of the deque and the processor start execution of the child thread. When a thread dies or stalls, a thread is popped from the back of the deque and execution of the popped thread is resumed. If a stalled thread becomes ready after an action the ready thread is pushed onto the back of the deque.

If a processor tries to pop a thread from its deque and the deque is empty, the processor tries to steal a thread from another processor. This is done by randomly selecting a processor to steal from and then steal a thread by popping a thread from the front of the victim's deque.

3.2.2 Prioritized Scheduling

The order in which actions are scheduled impacts the cache complexity. For P processors, a *prioritized schedule* is a schedule where at most P actions are scheduled at each step. A schedule is said to be *parallel* if $P > 1$ and *sequential* if $P = 1$. A P-schedule is *greedy* if at each step either P actions are scheduled or all ready actions are scheduled. Any greedy P-schedule of a DAG with work T_1 and depth T_∞ uses $T_P \leq T_1/P + T_\infty$ steps.

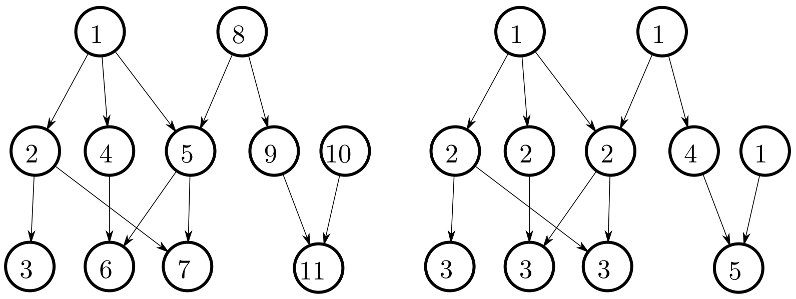
A P-schedule based on a sequential schedule, that determines the priority of the actions, attempts to deviate as little as possible from the sequential schedule. The goal of basing a P-schedule on a sequential schedule is to make as much progress on the sequential schedule as possible at each step, to obtain bounds as close to the bounds of the sequential schedule as possible.

A greedy P-schedule schedules the P ready actions with highest priority at each step. This makes the P-schedule work efficient[3]. Because the actions that are scheduled at a step only can be chosen from the ready actions and the schedule is greedy, the priority cannot always be followed strictly (see Figure 3.3). This means that some actions will be completed before they would be in the sequential schedule (see Figure 3.3). An *premature action* is an action the is completed before an action with lower priority [4].

The *DF-schedule* is a sequential schedule that schedules the actions in depth-first order. The *PDF-schedule* is the prioritized p-schedule based on the DF-schedule. A PDF-schedule and a DF-schedule is shown in Figure 3.3.

For a DAG $G(V, E)$, can the number of premature nodes in a P-schedule s_P, that is based on a 1-schedule S_1, be no more than $(P-1)\dot{d}epth(G)$. Any P-schedule S_P, based on an 1-schedule S_1 having M_1 cache misses on an ideal cache of size C_1 incurs M_1 cache misses on a shared ideal cache of size $C_1 + P \cdot depth(G)$.

The *shared ideal cache* is an expansion of the ideal cache model, where the cache is shared by a set of processors. that require that any cache line accessed by a



(a) sequential-depth-first schedule. (b) Parallel-depth-first schedule with three processors.

Figure 3.3: Example of a parallel-depth-first schedule base on the shown sequential-depth-first schedule. The circles represent tasks and are numbered according to the step they are scheduled in. The arrows represent data and spawn dependencies. Notice that some actions are complete prematurely in the Parallel-depth-first schedule, e.g. action with priority 8 in the 1DF-schedule.

premature node is never evicted from the cache.

We call an action that incurs a cache miss in S_P but not in S_1 *bad* and an action the incurs a cache miss in S_1 but not in S_P *good.* There is a one-to-one correspondence between good and bad actions [4]. For a Bad action v, it must be the case that v, by definition, incurs case hit in S_1 and a cache miss in S_P. Because v incurs a cache hit in S_1 there must exist an action v' that incurs a cache hit in S_1 and brings the cache line used by v into cache. v' must have lower priority the v as v' brings the data used by v into cache. v can only incur a cache miss in S_P if v is scheduled before v' in S_p. v is thus a premature action in S_P. If v is scheduled before v' in S_P then the data, must by definition of shared ideal cache, be in cache when v' is executed. v' must therefore be a good action. This proves that a prioritized schedule achieves the same cache bounds as the sequential schedule the parallel schedule is based on, under the assumption of the ideal cache model.

The DF-schedule achieves good cache bounds, as the PDF-schedule is based on the DF-schedule. Hence the PDF-schedule has near the same cache bound as the DF-schedule.

3.2.3 Parallel-Depth-First Scheduling

The *Parallel Depth First* (PDF) scheduler is a P-scheduler based on a sequential depth-first schedule. The PDF-scheduler schedules *tasks* instead of actions (defined in Section 3.1). The PDF-scheduler makes a simplifying assumption on the DAG that is scheduled. That any spawned children tasks are independent.

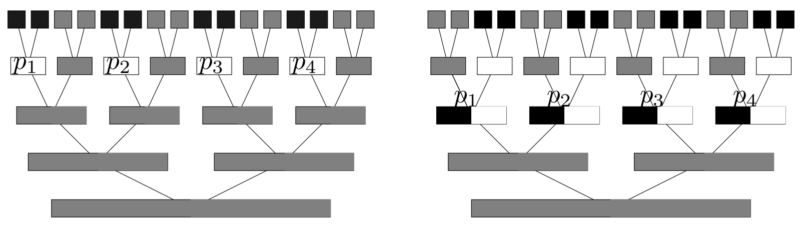
The PDF scheduler uses a central queue to store the ready tasks. All processors use this central queue to select which tasks to execute. The schedule is a prioritized schedule and the lowest priority tasks are scheduled before higher priority tasks. The task queue stores the tasks is sorted order by their priority. The priority is maintained in the queue by keeping the live tasks in the task queue.

When a task spawns a set of child tasks, the child tasks are ordered in some priority. The parent task is then replaced by the ordered set of children tasks in the task queue. When a child task is completed it is removed from the task queue and the last child task is replaced with the parent task.

The synchronization of tasks, used to detect when the execution of the parent task can be resumed, is done by associating synchronization counter with each task. When a child task dies the child task atomically add one to the synchronization counter of the parent task. If the counter is equal to the number of spawned child tasks, the child task is replaced with the parent task in the task queue.

3.2.4 Discussion

Work-stealing has the benefit that the ready deque is distributed to all processors. This lowers the contention that a central ready queue would introduce. Another benefit is that the cost of work balancing is only payed when work balancing is actually needed. The assumption of work stealing is that a child thread uses the same memory resources used by the parent thread. It is therefore beneficial to execute child threads on the same processor as the parent thread. This assumption it usually true and the benefit from keeping related threads on the same processor is highest when the processors do not share memory. The DAG is divided as early as possible, i.e. in a breath-first manner. Each processor then schedules its part of the DAG in a depth-first manner. The breath-first division of the work leads to bad cache complexity, as the amount of cooperative sharing is minimized.



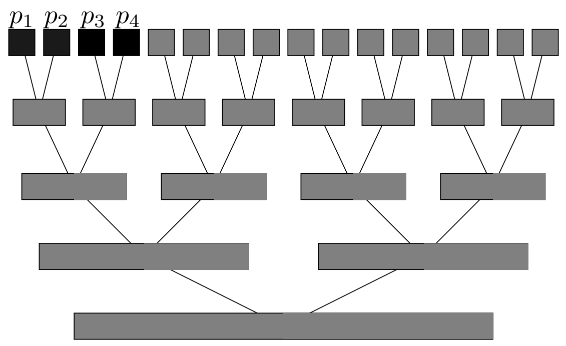
(a) Start of algorithm with work-stealing. (b) Later in the algorithm with work-stealing.

Figure 3.4: Example of a Work-Stealing scheduler execution of a parallel divide-and-conquer algorithm, on a CMP with a L2 cache that can contain half the input data. Each block represents a task, the length of a block coresponds to the amount of data accessed. Larger tasks consists of smaller subtasks (not shown) and can be done using more processors. A black task indicates that the data used in the task are not in cache and the task incur cache misses. The grey tasks indicate that the data used by the task is not in cache. A white task indicates that the data is already in cache.

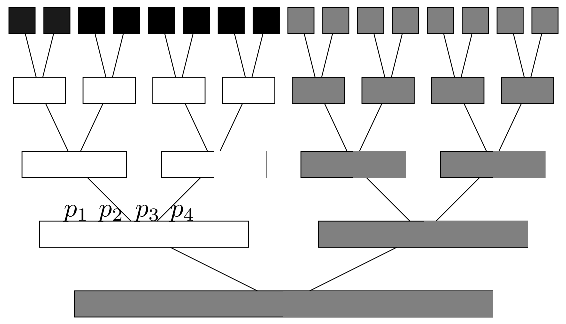
The most used online scheduling algorithm is work stealing. Work stealing is simple and efficient but work stealing is not always the best choice. The cache complexity of work stealing in the CMP-cache model is not optimal [3]. Chen et al. [9] shows that PDF-scheduling achieve better cache performance than

WS-scheduling on a system with one CMP.

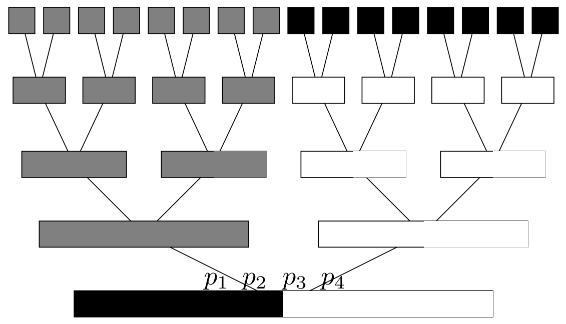
PDF-scheduling requires a central ready queue this can lead to a bottleneck on systems with many processors. The bottleneck from the central ready queue is not visible in the results from Chen et al. [9], as they only use a single CMP with a limited number of processors. When only a single CMP is used, the ready queue can then be optimized for the cache in the CMP model. The decrease in performance would be larger if the central queue is maintained by two or more processor chips.



(a) Start of the algorithm with parallel-depth-first scheduling.



(b) half way through the algorithm with parallel-depth-first scheduling.



(c) End of the algorithm with parallel-depth-first scheduling.

Figure 3.5: Example of a parallel-depth-first schedule of a divide-and-conquer algorithm, on a CMP with a L2 cache that can only hold half of the input data. Each block represents a task, the length of a block corresponds to the amount of data accessed. Larger tasks consist of smaller subtasks (not shown) and can be done using more processors. A black task indicate that the data used in the task are not in cache and the task incurs cache misses. The grey tasks indicates that the data used by the task is not in cache. A white task indicates that the data is already in cache.

4 Parallel Algorithms

In this Chapter some of the techniques and pitfalls in the construction of parallel algorithms are discussed. As a basis for the discussion and as examples, two different parallel algorithms for sorting are presented. Sorting was chosen as the example algorithmic problem, because it is non-trivial and has been studied intensively. The algorithms are analysed with respect to the CMP-cache model.

Two of the classical sequential-sorting algorithms, mergesort and quicksort, can, with care, be parallelized into efficient parallel sorting algorithms. The parallelizations of mergesort and quicksort presented here are simple and have good speedup while scaling to many processors. Neither of the two algorithms are work efficient, but work well in practice [21]. The mergesort algorithm described here is the parallel sorting algorithm implemented in the open-source library libstdc++ [20].

The mergesort algorithm described in this Chapter is based on a parallel algorithm for partitioning of multiple sorted sequences, that is presented by Varman et al. [23]. The quicksort algorithm described is based on a parallel pivot partitioning algorithm presented by Tsigas et al. [22].

4.1 Mergesort

Mergesort is commonly used for parallel sorting. The benefit of mergesort is the balanced division of the input data, which results in less work balancing. The disadvantage of mergesort is that the parallel merging is usually not done in-place, increasing the space bound of the sorting algorithm considerately.

4.1.1 A Naïve Parallelization

A naïve parallelization of mergesort would be to divide the input data into two runs and to recurse on the runs in parallel, and to merge the runs by using the ordinary sequential merge routine. The work of the naïve merge sort ($\mathit{work}(n)$) can be described by the recurrence [11, Section 27.3]:

$$\begin{aligned}\mathit{work}(n) &= 2\mathit{work}(n/2) + \mathcal{O}(n) \\ &= \mathcal{O}(n \lg n)\end{aligned}$$

In the naïve parallel mergesort, the two recursions are run in parallel. The depth can thus be described by the following recurrence [11, Section 27.3]

$$\begin{aligned} depth(n) &= depth(n/2) + \mathcal{O}(n) \\ &= \mathcal{O}(n) \end{aligned}$$

The maximal possible speedup of the naïve parallelization of mergesort is thus [11, section 27.3]:

$$Speedup_{\infty} = \frac{work(n)}{depth(n)} = \lg n$$

This naïve parallelization has very limited speedup, due to the use of the sequential merging routine. As the runs are merged fewer and fewer merges can be performed in parallel, at the end a single processor must merge the whole input. This is an example of Amdahl's law (see Section 3.1.1) and illustrates why it is important that as much as possible of the algorithm is parallelized.

4.1.2 An Efficient Parallelization

To effectively parallelize mergesort a parallel merge procedure is needed. We would specifically like to have a parallel multi-way merge, as multi-way merge preform less scans over the data (see Section 2.1). The actual merging in multi-way merge is trivial to parallelize, each processor just merge $1/P$ (recall that P is then number of processors). The algorithmic interesting part of parallel multi-way merging is finding the index in each of the sequences to be merged where processor p should start merging from. The case for the first processor is trivial, it merges form the beginning of each sequence. The other processors partitions the sequences into two parts, the elements sorted by the previous processors and the not yet merged part. The size of the two partitions are known before the partitions is found, as the number of elements that are each processor is fixed. This type partitioning is called an f-way partition, where f is the fraction of elements in the first partition set.

Definition 4.1.1. *Let S be a set of elements, divided into m sequences, with length n and each sorted in increasing order. Let f be a fraction, $0 < f \leq 1$. An f-way partition of S is a partition of S into two parts, L and H, such that the following two properties are fulfilled:*

Domination criterion: *H dominates L, that is all elements in L is smaller than all elements in H.*

Size criterion: *L contains $|L| = \lceil f|S| \rceil$ elements.*

We can obtain an efficient parallel multiway merge algorithm. Given P processors (numbered from 0 to $P-1$) and the set of sequences S, assumed to have the same length m, we construct a parallel multiway merge algorithm as follows. Processor P_p first finds the p/P-way partition of S. Once the p/P-way partition has been found, each processor starts merging using the sequential multi-way merging algorithm until it has merged $|s|m/P$ elements. The P merged sequences can the be concatenated, in the numerical order by index of the processor that merged the sequences, to obtain the merged sequence. This concatenation can be done by during the merging by letting each processor merge directly into the correct position in the output array. Because the f-way partitioning can be run in parallel by P processors, the actual f-way partitioning algorithm can be sequential without any performance degradation.

4.1.3 f-Way Partitioning

The f-way partitioning algorithm described here uses an iterative approach to obtain a partition of the m sorted sequences. At each iteration a subsection of the elements in S is called the *sample.* The sample considered at iteration k is denoted $S(k)$ and the f-way partition of $S(k)$ is the two sets $L(k)$ and $H(k)$. The algorithm starts with a small sample and at each iteration, the number of elements in the sample is doubled, while maintaining the f-way partition.

We make some simplifying assumptions to ease the description of the algorithm. These assumptions are discussed later in Section 4.1.4.

- The sequences are all of equal length n
- The length of the sequences is $2^r - 1$ for a constant integer r

Let $S^i = a(i,1), a(i,2), \dots, a(i,n), 0 \leq i \leq m$ denote the ith sequence. Let $w_k = 2^{r-k-1}, 0 \leq k \leq r-1$. The sample considered at iteration k is then every w_kth element of each sequence. Note that w_k is halved at each iteration and the size of the sample is thus doubled. We denote the sample from the ith sequence considered at iteration k as $S^i(k)$.

The partition of $S(k)$ into $L(k)$ and $H(k)$ is defined by a *boundary* $B(k)$ and the largest element in $L(k)$, denoted l_{max}. The algorithm thus maintains the boundary and l_{max} during the iterations.

Definition 4.1.2. *The boundary B is the ordered set of the indices of the largest elements from each sequence S^i that is a member of L . The boundary for a sequence $S^i(k)$ is thus defined as:*

$$b^i(k) = \begin{cases} 0 & \text{if } L(k) \cap S^i(k) = \emptyset, \\ max\{j : a(i,j) \in L(k) \cap S^i(k)\} & \text{otherwise.} \end{cases}$$

The boundary $b^i(k)$ for sequence i is the index of the largest element in the partition set $L(k)$. The boundary thus define the index where the partition set $L(k)$ and $H(k)$ begin. All elements in $S^i(k)$ with index less than $b^i(k)$ is in $L(k)$ and all elements with index larger than $b^i(k)$ are in $H(k)$. After the last iteration the boundary define the f-way partition of S.

Each iteration consist of three steps;

1. Double the size of the sample
2. Reestablish the boundary based on l_{max}
3. Reestablish the size criterion

At the beginning of an iteration k, the size of the sample is doubled. The boundary from the previous iteration will have assigned some of the elements, that were included in the sample in this iteration. All elements in $S^i(k)$ included in iteration k with an index that is less than $b^i(k-1)$ are in $L(k)$ and thus decided. The element included in $S^i(k)$, in iteration k, that has the index just after $b^i(k)$ are undecided. All elements included in $S^i(k)$ in iteration k that have an index larger than $b^i(k)$ are in $H(k)$ and thus decided.

Each undecided element u^i is compared to l_{max} and is assigned to a partition set. If u^i is smaller than l_{max}, u^i is assigned to $L(k)$ by updating the boundary $b^i(k)$ to the element index of u^i. If u^i is larger than l_{max} then u^i is assigned to $H(k)$ and the boundary is not changed.

When the boundary has been reestablished, the size criterion may be violated. To reestablish the size criterion a number of elements must be moved from one partition set to the other. If $L(k)$ is too large, then the largest elements are moved from $L(k)$ to $H(k)$. If $L(k)$ is too small, then smallest elements in $H(k)$ are moved from $H(k)$ to $L(k)$. This reestablishes the size criterion. The partition of $S(k)$ is now found and the next iteration can be started.

4.1.4 Example

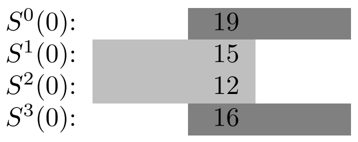
Figure 4.0 show an example of the f-way partition algorithm. The algorithm is run with four sequences each with 7 elements r is thus 3 (recall the assumption that each sequence has length $2^r - 1$. Figure 4.1b show the basis partition. The boundary is $b^1(0) = 3$, $b^(0) = 3$. $b^0(0)$ and $b^3(0)$ are undefined. l_{max} is 15.

Figure 4.1c show the first step in iteration one; the sample size have been doubled. Exactly four elements are undecided. The other elements are decided by the boundary from the basis. Each undecided element are compared to l_{max} and assigned to a partition set. The resulting partition is shown in Figure 4.1d. The size criterion is fulfilled and iteration is complete.

Figure 4.1e show the first step in iteration two, where the sample size have been doubled. Exactly four elements are undecided. Figure 4.0f show the boundary after it have been reestablished. The size criterion is not fulfilled as $|L(2)| = 15$ and $|H(2)| = 13$; $L(2)$ contain one element too many. The largest element in $L(2)$ is found (15) and the boundary is adjusted to transfer this element to $H(2)$, $b^1(2)$ is set to 2. The resulting boundary is shown in Figure 4.0g.

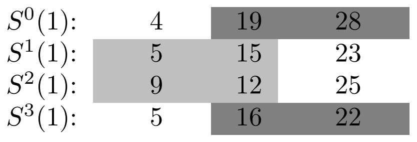
S^0:	2	4	8	19	24	28	30
S^1:	3	6	14	15	17	23	27
S^2:	2	9	11	12	13	25	29
S^3:	1	5	13	16	21	22	23

(a) Four sequences that will be used for an example run of the partitioning algorithm.



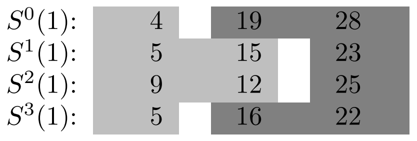
$S^0(0)$:	19
$S^1(0)$:	15
$S^2(0)$:	12
$S^3(0)$:	16

(b) The basis partition



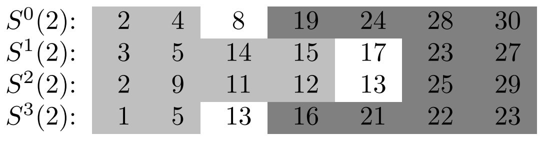
$S^0(1)$:	4	19	28
$S^1(1)$:	5	15	23
$S^2(1)$:	9	12	25
$S^3(1)$:	5	16	22

(c) Beginning of iteration one.



$S^0(1)$:	4	19	28
$S^1(1)$:	5	15	23
$S^2(1)$:	9	12	25
$S^3(1)$:	5	16	22

(d) Reestablishing the boundary in iteration one.



$S^0(2)$:	2	4	8	19	24	28	30
$S^1(2)$:	3	5	14	15	17	23	27
$S^2(2)$:	2	9	11	12	13	25	29
$S^3(2)$:	1	5	13	16	21	22	23

(e) Beginning of iteration two, the size of the sample is doubled.

$S^0(2)$:	2	4	8	19	24	28	30
$S^1(2)$:	3	5	14	15	17	23	27
$S^2(2)$:	2	9	11	12	13	25	29
$S^3(2)$:	1	5	13	16	21	22	23

(f) Iteration two: The undecided elements have been assigned to a partition set according to l_{max}.

$S^0(2)$:	2	4	8	19	24	28	30
$S^1(2)$:	3	5	14	15	17	23	27
$S^2(1)$:	2	9	11	12	13	25	29
$S^3(2)$:	1	5	13	16	21	22	23

(g) Iteration two: The resulting boundary after the size criterion have been reestablished.

Figure 4.0: An example run of the partitioning algorithm on four sequences with $f = \frac{1}{2}$. (b) show the four sorted sequences that are partitioned. (b) show the basis partition. The dark grey blocks are the elements in the partition set $H(k)$ and the light grey blocks are the elements in $L(k)$. The elements with white background are undecided.

Assumptions and Basis

The assumptions made in the description of the partitioning algorithm were that all the sequences are of equal length and that the length of the sequences is $n = 2^r - 1$ for a positive integer r.

The general solution for sequences of unequal length is not discussed here. As it, due to the large increase in the length of the sequences, is more theoretical than practical [23]. When the partitioning algorithm is used for merging in merge-sort the sequences are of equal length, except possibly a single sequence that will contain fewer elements. This smaller sequence is only present if the input elements cannot divided evenly into runs. The smaller sequence can conceptually be padded with $+\infty$ elements so that it has the same length as the rest of the sequence. The algorithm can thus be used even if this smaller sequence is present.

We can remove the assumption that the sequences have length of $n = 2^r - 1$ for a positive integer r, by constructing the basis as follows. We first define β as a positive integer for which $2^{\beta-1} < fn \leq 2^{\beta}$. The basis for the induction is constructed by selecting every 2^{β}th element of each sequence. The number of elements in the basis, per sequence, is thus $\alpha = \left\lfloor \frac{n}{2^{\beta}} \right\rfloor$. The basis is then the f-partition of the αm sample elements. Because of the way β is chosen the basis will contain m elements and the basis can be found with a simple tournament tree in $\mathcal{O}(m \log m)$ comparisons.

Theorem 4.1.1. *The number of elements in the basis sample is m. This comes directly from the definition of β.*

Proof.

$$f|S| \leq 2^{\beta}$$

Using that the sequences are of equal length and thus $|S| = nm$, we get:

$$\frac{1}{2^\beta} \leq \frac{m}{f|S|}$$

By the definition of α and $|S| = nm$, we get:

$$\alpha \leq \left\lfloor \frac{nm}{fnm} \right\rfloor$$
$$f\alpha \leq \left\lfloor \frac{nm}{nm} \right\rfloor = 1$$

The size of the basis sample is thus:

$$f\alpha m \leq m$$

□

Complexity Analysis

With a basis constructed as described above, the partitioning algorithm uses $r = \mathcal{O}(\lg fn)$ iterations. Each iteration uses $\mathcal{O}(m)$ comparisons for reestablishing the boundary. The size criterion can be violated by at most m elements and reestablishing can done, with a tournament tree, using at most $m \lg m$ element comparisons. The algorithm thus uses $\mathcal{O}(m \lg m \cdot \lg fn)$ element comparisons [23].

At each iteration one new element per sequence is used. This use of a new element causes one I/O and incurs one cache miss. The algorithm thus has an I/O and cache complexity of $\mathcal{O}(m \log fn)$. Note that the simultaneous execution of the algorithm on a CMP does not increase the cache complexity as the elements accessed at each iteration will be the same for all processors and an element will therefore only incur a single cache-miss for all processors.

4.1.5 Parallel Multiway Mergesort

We can now construct a parallel multiway merge algorithm as described in Section 4.1.2. This parallel multiway merge algorithm can then be used to construct a parallel multiway mergesort algorithm. First we divide the input into runs that can 'fit' in main memory. The runs are then sorted and the results merged using the above parallel algorithm.

Varman et. al. [23] and Singler et. al. [21] both describe a mergesort algorithm, where runs of size N/P are sorted in parallel by an I/O optimal sequential algorithm. The runs are then merged as described above. This is neither cache nor I/O effective. The sorting of all runs in parallel causes extra cache misses (in the CMP-cache model). The I/O performance is not optimal either as the sorting the runs in parallel. In the CMP-cache 4model this will cause the processors to uses the same main memory and the sorting of a run can thus not be done in memory, incurring degradation in the I/O performance as well as further increasing the number of cache misses.

The sorting of the runs must be done in a way that optimizes the cache and I/O performance while still being scalable. The run size must optimize I/O, cache performance, and scalability. A simple way would be to use a run size of $\mathcal{O}(M/P)$ and then sort P runs in parallel followed by a parallel multiway merge of the sorted runs. This improves the I/O performance while still having high scalability, but caching would be purely competitive (see sec. 2.3). To improve the cache performance, we must perform the sorting of a single run at a time, in parallel on all P processors. This means that the run size can be chosen as $\mathcal{O}(M)$ but only one run can be sorted at a time. This means that the multiway mergesort algorithm depends on another parallel sorting algorithm to sort the runs, just as the sequential multiway mergesort uses another sequential sorting algorithm to sort the runs. Sorting one run at a time limits the scalability and the algorithm cannot utilize as many processors as would be possible if all the runs were sorted in parallel with a parallel sorting algorithm.

4.2 Quicksort

Quicksort is a well-known sequential sorting algorithm. The worst-case running time is non-optimal ($\mathcal{O}(n^2)$), due to the randomization used, but the average running time is optimal. Quicksort is a marriage-before-conquest algorithm, which is part of the divide-and-conquer paradigm. The input is first divided, then the marriage of the solutions to the two subproblems is found and then the two subproblems are solved. This corresponds to a mergesort where the input is merged and the runs then sorted. Quicksort works by picking a pivot element and partitioning the input sequence into two parts L and H. The partitioning is done in such that any element in L is smaller than or equal to the pivot element and any element in H is larger than the pivot element. The input is recursively partitioned, until each partition is of a certain size, the partition is then sorted directly. To obtain a reasonable speedup it is necessary to find a way to partition the input array in parallel, as it was the case with mergesort.

4.2.1 Parallel Pivot Partitioning

The parallel partitioning algorithm below is based on the work of [22] and is a parallelization of a sequential partition algorithm. The sequential partition algorithm first picks a pivot value and then scans the array from both ends. The forward scan from the beginning of the sequence proceed until an element that is higher than the pivot value is found. The backwards scan from the end of the sequence proceed until an element that is smaller than or equal to the pivot is found. The two elements found by the scans are then swapped. When the scans meet the sequence is partitioned. The parallel partitioning algorithm consists of two phases, a parallel partition phase and the sequential partition phase. The parallel phase does most of the work and the sequential phase does the remaining work that could not be done in parallel.

Parallel Partition Phase

The sequence is divided into N blocks of b consecutive elements. We assume that the input divides into blocks, this assumption is trivially handled in practice. The goal of the parallel partitioning phase is then to *neutralize* as many blocks as possible. A block is said to be L-neutralized if it contains no elements larger than the pivot element. A block is likewise said to be H-neutralized if it contains no elements smaller that or equal to the pivot element. Each processor p picks one block l from the L side and one block h from the H side of the input array. The picked blocks are marked in an atomic way to prevent any blocks from being picked more than once.

The processor then neutralizes the blocks by swapping elements between the blocks, as in the sequential algorithm. Elements in l that are larger than the pivot are swapped with elements in h that are smaller than or equal to the pivot. When either l or h has been neutralized, a new block from the corresponding side of the input is picked and the neutralization continues until all blocks have been picked by a processor. Each processor can then have at most one unneutralized block remaining. These unneutralized blocks are put into an array of remaining blocks. The remaining blocks are neutralized in the sequential phase.

Each processor counts how many blocks it has L-neutralized and how many it have H-neutralized. The number of neutralized blocks for each processor are added to get the total number of L-neutralized blocks N_L, and the total number of H-neutralized blocks, N_H. The numbers of neutralized blocks divide the array into three parts; a low part $[0, N_L]$, a middle part $[N_L + 1, N - N_H - 1]$, and a high part $[N - N_H, N]$. Any neutralized block in the left or the right part will be positioned correctly in relation to the pivot, but both sides can contain unneutralized blocks. The blocks in the middle part can be neutralized to any side or can be unneutralized.

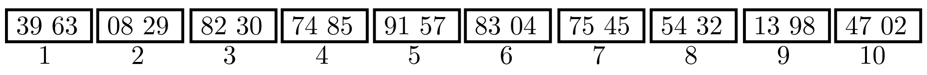
Sequential Partition Phase

The sequential partitioning phase starts when parallel partition phase is complete. The sequential partition phase finishes the partitioning that was begun in the parallel partitioning phase; neutralizing the remaining blocks and moving neutralized blocks to the correct position with respect to the pivot. An unneutralized block can reside in any of the three partitions. The neutralized blocks in the low part and the high part are correctly positioned. Any neutralized blocks in the middle part must be moved to the correct position.

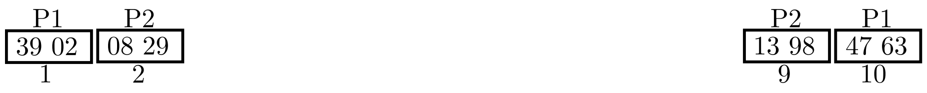
The remaining blocks, that was not neutralized in the parallel phase, is first swapped into the middle part of the sequence. The middle part can then be partitioned sequentially.

An Example

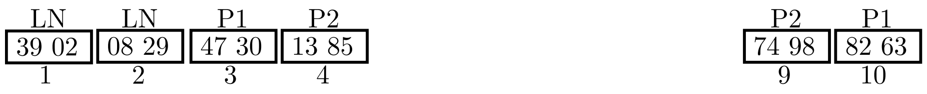
Figure 4.1 shows an example of the parallel partitioning algorithm.



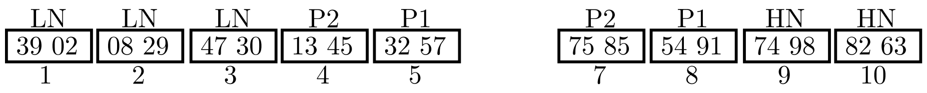
(a) The example input sequence. The sequence has been divided into blocks of two elements. The blocks are numbered, shown below each block. The pivot value has been chosen as 60



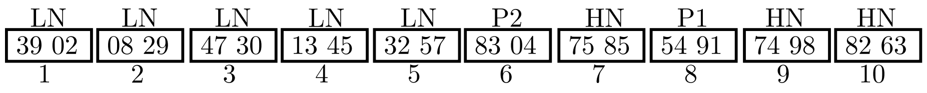
(b) Processor one picks block one as l and block ten as h. Processor two picks block two as l and block nine as h. Processor one swaps 02 and 63



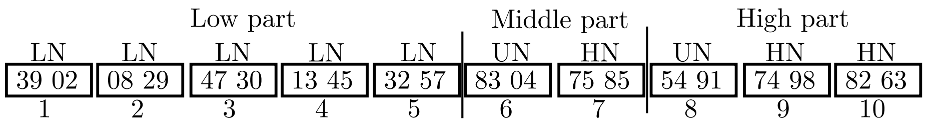
(c) Blocks one and two have been L-neutralized. Processor one picks block three as l and processor two picks block four as l. Processor one swaps 82 and 47 and processor two swaps 74 and 13.



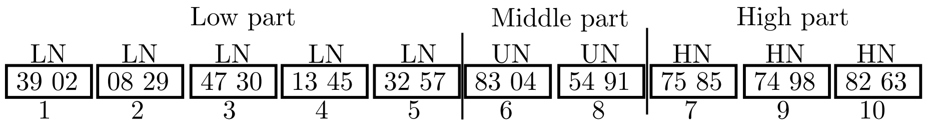
(d) Blocks nine and ten have been H-neutralized and block three has been l-neutralized. Processor one picks block eight as h and block five as l. Processor two picks block seven as h. Processor one swaps 91 and 32. Processor two swaps 85 and 45



(e) Blocks four and five have been L-neutralized and block seven has4 been H-neutralized. Processor two picks block six as l. Both processors only have one block and no elements can be swapped.



(f) Result of parallel partitioning phase. Blocks six and eight are unneutralized. The low, middle, and high part is shown, separated by vertical lines



(g) The sequential partitioning phase first swap block eight seven. The middle part are then partitioned by swapping Elements 83 and 54 are swapped.

Figure 4.1: An example of the parallel partitioning algorithm run by two processors.

Analysis

The parallel partitioning phase does $\mathcal{O}(n)$ work using $\mathcal{O}\left(\frac{n}{P}\right)$ time, where n is the number of elements. The sequential partitioning phase takes $\mathcal{O}(P \cdot (b))$ time neutralizing the P blocks (recall that b is the block size). The total time used by the partition algorithm is thus $\mathcal{O}\left(\frac{n}{P} + Pb\right)$. If $b << N$ and $P << \frac{n}{2b}$ then the algorithm will have linear speed up. This breaks down when the number of processors grows as the sequential phases will use more and more time. The maximal number of processors that the algorithm can use is $P = \mathcal{O}\left(\frac{n}{2b}\right)$, then all blocks are picked by a processor at once and half of the blocks must, in worst-case, be neutralized in the sequential phase.

The parallel partitioning phase is a parallel scan, if $2P$ blocks can fit in main memory. The number of I/Os is then $\mathcal{O}(N/B)$. The cache behaviour of the parallel partition phase is a scan over both the left and the right block. The number of cache misses is therefore $\mathcal{O}(b/l_1)$ where l_1 is the cache line size of L1 cache. The L2 cache complexity is similarly $\mathcal{O}(b/l_2)$, where l_2 is the L2 cache line size. Tsigas et al. [22] limit the block size, so that two array blocks can fit in L1 cache at once, that is $b \leq c_1/2$. Theoretically this is not necessary as the access is a scan, and the number of cache misses is optimal for any block size. It is possible that choosing the block size such that two block can fit in L1 cache is faster as it will allow more of the blocks to be prefetched by the processor and it might allow better branch prediction. It is necessary to experiment with this to reach a conclusion.

The sequential partitioning phase performs no I/Os if the P middle blocks can fit in main memory. The P unneutralized array blocks are the last used by the parallel partition phases and will thus be in main memory. The cache complexity of the sequential partitioning phase is similar to the I/O. If P blocks can fit in L2 cache then only the left out elements incur cache misses. The P array blocks needed by the sequential phase will already be in L2 cache and the sequential partition phase will incur no L2 cache misses.

Singler et al. [21] propose to recursively use the parallel partitioning phase. Each time with half the number of processors. This could result in an increase of speedup if many processors are used. But the recursion should probably be stopped at some constant number of processors to limit the depth.

The partitioning algorithm has four things against it. Firstly the scalability and speedup limit each other; if many processors are used the speedup degrades. Secondly, the array is only divided into two parts, which will result in an I/O inefficient quicksort algorithm. Thirdly, the algorithm contains a sequential part and as we know from Amdahl's law this should be avoided. Lastly, there is not an efficient way of selecting a pivot that guaranties a balanced partitioning of the input. This requires the use of a work balancing techniques.

4.2.2 Parallel Quicksort

Tsigas et al. [22] described a parallel quicksort using the above parallel partitioning algorithm. First a pivot value is selected. This is done with the median of three strategy, where the median of the first, the last and the middle element is selected as pivot. The input array is then partitioned using the above parallel partition algorithm. When the input has been partitioned, the processors are partitioned and assigned to a partition. The processors are partitioned so that the number of processors in a partition is proportional to the size of the array partition they are assigned to. This is done recursively, until each array partition has a single processor assigned. When only a single processor is assigned to a partition, the processor begin a sequential sorting phase. The processor keeps partitioning the with a sequential partitioning algorithm. When a partition is small enough to fit in cache, the partition is sorted with a sequential sorting algorithm e.g. insertion sort.

The partitioning is not guarantied to be balanced and some work balancing is required to insure that processors that were given a small partition can help processors that were given a large partition. To allow another processor to help sort an input partition, each partition is spawned as a task that can be run in parallel. Tsigas et al. [22] use the work-stealing algorithm to achieve the work balancing. The scheduled tasks thus correspond to an partition that either should be partition into smaller parts, spawning more tasks, or a small partition that can be sorted directly.

The parallel quicksort cannot reach an optimal I/O performance as the partitioning algorithm only divides the input into two parts. The number of scans over the input is thus $\mathcal{O}(\log n)$ which is non-optimal. The cache performance of the algorithm described by Tsigas et al. [22] can be improved by letting all processors work on one of the input partitions instead of dividing them across both. When a partition is small enough to fit in L1 cache, it can be sorted directly. This corresponds to using the PDF scheduling algorithm instead of work stealing.4

4.3 Discussion of Parallel Algorithms

The two described parallel sorting algorithms have different trade-offs. The parallel mergesort algorithm use double the needed space, because the merge algorithm is not in-place. The extra space used could be limited to a single extra array, by alternating between one array used for input and the other used for output. The parallel quicksort algorithm is not stable as the partitioning algorithm is not stable. The mergesort algorithm has good I/O bounds but the cache performance and the speedup limit each other. The quicksort algorithm has good cache performance, if the PDF scheduler is used, but the I/O performance is not optimal. It seems natural to try to combine the two by using the quicksort algorithm to sort the mergesort runs. This combination will however result in a sorting algorithm that is neither stable nor in-place.

Neither of the algorithms are cache oblivious and the question arises whether a stable, in-place, I/O optimal, cache oblivious, work optimal, parallel sorting algorithm exists. Cole and Ramachandran [10] presented a work optimal and cache oblivious algorithm, but it is more complicated than the presented algorithms and whether it has practically usable is yet to be determined.

The above algorithms are meant as examples of parallel algorithms. One of the most basic operations in parallel algorithms is the division of data in order to allow computation to run in parallel on the smaller parts. We can also see that it results in better I/O performance if the input can be divided directly instead of recursively. The parallel computation should be done on small parts in order to maximize the number of processors the algorithms can scale to. We also observe that processors should work as long as possible on the same data parts, to maximize the cache performance.

Nested parallelism allows programmers to ignore the implementation of a parallel algorithm and to use it just as one would in a sequential setting, without any degradation of scalability or speedup. This technique allows a problem to be broken into subproblems that can be solved with an off-the-shelf algorithm. It is then the programmers job to identify the subproblems that can be run in parallel and communicate this to the programming library.

5 Related Work

Concurrent execution of a program requires that the program can be decomposed into parts that can be run concurrently. There are two general ways of decomposing a program into these parts. If the program consists of independent tasks, these can be run in parallel. This is called task decomposition as the focus is on the tasks. The other way to decompose a program is by focusing on the data used by the program. Often the data used by a program can be divided into smaller problems, each represented by subsets of the data. These subproblems can then be solved in parallel and the solution combined to obtain the solution to the original problem. This technique is similar to the divide-and-conquer paradigm. A library should support both types of decompositions to make as much parallelism as possible available to the programmer. Data decomposition can be modelled as a task decomposition by spawning the same task several times with different input data.

5.1 Granularity

Granularity is an important factor in the performance of parallel code execution. Libraries for parallel programming introduce two new performance factors.

Available parallelism: if the amount of parallelism expressed by the user is too low, then processors will idle leading to poor performance.

Scheduler overhead: If too much parallelism is expressed by the user, the overhead introduced by the scheduler becomes too large and the performance drops.

To obtain optimal performance from a parallel library, the user has to balance the above two factors. This requires that the user writes tasks that are neither too larger nor too small. The optimal task size depends on many factors such as the number of processors, cache layout, cache sizes, and the scheduler used. The granularity must therefore be found experimentally for a specific program and system.

The granularity available due to scheduler overhead limits the scalability to less than it would be with very fine granularity. This limits the number of practically usable processors, usually to less than the theoretical usable number of processors.

5.2 Other Libraries

This Section describes how other libraries for parallel programming have been designed. The next Section discusses differences and similarities of the libraries. Describing all the available libraries is beyond the scope of this thesis. The libraries described here are only a small part of the existing libraries, but the libraries have been chosen to highlight some of the trends in the design of such libraries.

5.2.1 Cilk

Cilk is an extension to C by Frigo et al. [16], made for symmetric multiprocessing, where a collection of identical processors are used. The idea of Cilk is to have the programmer expose the available parallelism in the program and then let Cilk handle the actual execution of the program.

Cilk introduces the keywords `spawn` and `sync`. The `spawn` keyword creates a new thread of execution that may run in parallel with the spawning thread. The `spawn` statement can `spawn` special Cilk-procedures or unnamed functions in the form of a code block. The Cilk-procedures allow nested parallelism[16].

To synchronize two threads the keyword `sync` is used. The `sync` keyword blocks the parent thread until the child thread has finished. The child thread then dies and the parent thread continues after the sync. A `spawn` may return a value. The grammar of a `spawn` statement takes the form `[`*`lhs op`*`]` **`spawn`** $foo(arg_1, arg_2, \ldots arg_n)$, where foo is a Cilk procedure. The so called *Inlet functions* allow more complex actions on `spawn` returns. An inlet function is invoked when a `spawn` returns. To avoid race conditions inlet functions are atomic[16].

Cilk contains *hyperobjects* that help the user avoid race conditions[14]. Hyperobjects are an encapsulation of data that removes the synchronisation responsibility from the user by doing it automatically. A hyperobject present a view local to a thread of a global object. The views from parent and child thread are combined when the two threads `sync`. In [14] three hyperobjects are presented:

Reducer: reduces two views with a specified function. An example could be an integer sum reducer, where each thread has a local integer as the view and the reduction function is integer addition.

Holder: generalizes thread local storage. If a thread uses a global variable and creates two separate threads of execution by spawning, a holder hyperobject allows each thread to treat the global variable as thread local without risk of race conditions occurring.

Splitter: is a special case of a holder hyper object where, when the thread dies, the value of the date in the hyper object must be restored to its value before the spawn.

Hyperobjects in Cilk simplifies the use of thread local data by allowing global data to be viewed as local.

5.2.2 Standard Template Adaptive Parallel Library

The Standard Template Adaptive Parallel Library (STAPL), presented by Buss et al. [7], is a parallelization of the STL, made for distributed systems. STAPL consists of three major components: `pContainers`, `pViews`, and `pAlgorithms`. pContainers are data structures that allow concurrent operation on the structure, e.g. simultaneous inserts and deletes. The individual operations on the data structure itself may be parallelized by the pContainer. pAlgorithms are parallelizations of the STL algorithms. The pAlgorithms both provide the STL interface and a STAPL interface that uses pViews. pAlgorithms are expressed as task graphs. The vertices in a task graph are the individual tasks and the edges are data dependencies. STAPL runtime system uses an executor to find and execute the ready tasks.

pViews, presented by Buss et al. [6], are used to abstract the underlying data structure, just as iterators are in the STL. pViews make data decomposition easier and more general, by allowing algorithms to work with pViews instead of the underlying data structures. A pView consists of a base pContainer and a function mapping elements in the pContainer to elements in the pView. This mapping function allows pViews to express more complex part of a pContainer than simple sequences. A pView can represent every second element or a translation of a array of integers to an array of floats. Where as STL iterators provide functions for transversing a data structure, pViews provide functions with more focus on parallelism. A pView can be used to provide a balanced division of a container into a number of subparts, without actually dividing the pContainer. A pView could also be used to provide a lazy transformation of the elements in a pContainer without necessarily changing the pContainer. pViews are also used to manipulate sets of elements in pContainers, providing a clean interface for bulk operations.

Using pViews to perform data decomposition is beneficial. This moves the act of decomposing a range into subranges from the algorithm to the data structure. The STAPL uses this to provide a way to provide distributed data structures without forcing the user to distinguish between local and remote parts of the data structure. The same technique can be used in a shared memory library to let the data structure decide the best way to perform the decomposition based on the cache layout of the data structure[6].

5.2.3 Threading building blocks

Threading building blocks (TBB) is a large library made by Intel [17] . TBB is a pure c++ framework optimized for use on a single CMP and uses the work-stealing scheduler. TBB use a task functor class to allow nested parallelism. Any number of tasks can be spawned and TBB allows the user to `spawn` a list of tasks to support spawning of a dynamic number of tasks. Spawning a list of tasks result in all the tasks in the list being inserted into the ready pool, if the list is large this can cause a bottleneck as the tasks are stolen individually by work stealing scheduler, it therefore recommended that recursive spawning is used instead. The task class in TBB defines the task's execution function to return the next task the thread should execute, returning `NULL` causes the scheduler to automatically select the next task to execute. The task execution function is chosen as `execute()` instead of the traditionally used `operator()`.

To make data decomposition easier a range concept is included in TBB. A range can be split into two parts and a function `is_divisible` is used to decide if a range is large enough to be split. This is used in the library to recursively split ranges into parts that can be used as input to a task. The `is_divisible` function is used to control the task granularity.

TBB defines a set of generic algorithm skeletons that can be used to work with tasks, including `parallel_for`, `parallel_reduce` and `parallel_scan`[1]. These algorithms can be used with the range concept to control the task granularity. The body of the algorithms must be defined as a functor. The algorithm divides the input range recursively until the subranges cannot be divide more, `operator()` is then called with the range as input. Because the task class uses the `execute()` function to execute a task, tasks can be used in combination with the algorithms, but the user must handle all argument passing and no value can be returned directly.

TBB also has a library of containers that can be used concurrently. The containers can be used as a range allowing the generic algorithms in the library to work on an abstract range. This makes it possible to use the library algorithms with the data structures in the library.

5.2.4 Multi-Core Standard Template Library

The Multi-Core Standard Template Library (MCSTL) is a parallelization of the STL algorithms presented by Singler et al. [21]. The Goal of MCSTL is to allow easy parallelization by replacing STL algorithms with their parallel counter parts in MCSTL. One problem is that the parallel algorithm does not has exactly the same behaviour the STL version have. As an example merge is mentioned in, the STL merge allows the two sequences to overlap, but the parallel version fails if this is the case.

The MCSTL demonstrates that it is possible to implement algorithms that are both efficient and general enough that the reuseability known from sequential code is possible. It is also possible to nest the algorithms, this is used internally in the library but is also available to the user. The algorithms that are used in the implementation of the MCSTL also use the results from the external memory model. This mixture of models are very successful. The authors of the MCSTL notes that the external memory library STXXL is limited by computation bounds and not I/O [21].

The MCSTL does not make it possible for the user to implement additional parallel algorithms. From Amdahl's law we know that as little as possible of the code should be limited to sequential execution. A library should therefore allow the user to implement parallel algorithms that can be used just as the algorithms supplied by the library, as it is the case in STAPL.

[1]The `parallel_scan` algorithm skeleton in TBB use the parallel prefix meaning of scan and not the scan concept used in external memory model(see Section 2.1).

5.3 Interface Design

Simplicity of algorithms is important in both sequential and parallel computing. In parallel computing algorithms can easily get complicated and the special cases can take up more programming time than the base algorithm. Nested parallelism (see Section 3.1) allows any task to `spawn` more tasks that can be run in parallel. Nested parallelism allows parallel algorithms to be used just as we would a sequential algorithm. An algorithm can thus be divided into sub-problems and a parallel algorithm can be used to solve each sub-problem. Nesting of parallel algorithms must be provided in a transparent manner that allows a parallel algorithm to be used just as if it were a sequential algorithm.

In the libraries presented above are the interfaces differences of Cilk and TBB interesting. Cilk has a very simple interface, partly because of the use of a precompilier, but the interface of Cilk is limited to only a few functions. This makes Cilk very simple to learn, but the user is forced to use recursive spawning to `spawn` a dynamic number of tasks. TBB has a very large interface and the generic algorithms are used more than direct calls to spawn. The task class in TBB are quite complex and allocating and deallocating a task object must be done with special library function. This design discourages users from using nested parallelism. The interface should optimally be as simple and intuitive as Cilk's but still letting the user avoid manually granularity control and manual argument and return value management.

Even though TBB offers a wide selection of generic algorithm skeletons more concrete generic algorithms, which are available in the MCSTL, are missing. TBB only includes a sorting algorithm. The user is required to either self implement these algorithms them self or find third party implementation. Having more generic algorithms and fewer algorithm skeletons would be desirable, especially if combined with a simpler interface.

5.4 Views

TBB uses the range concept and STAPL uses views, to make data decomposition easier for the user. The STAPL view goes further than TBB's range concept and allows a view to change the way data is presented. The TBB range concept makes it easy for the user to recursively divide data into parts of a certain size. The STAPL balanced view allows the user to divide a data structure into blocks of similar size. The strength of both the range and the view concept is that it allows construction of generic algorithms, by using a range or a view in the same way the standard template library use iterators to abstract way the actual data structure. The concepts can thus be thought of as a parallel iterator.

Views have been proposed for sequential programming [25] as a combination of algorithms and iterators. In parallel programming they are practical as it is easier to divide a view than a range of two iterators. The interface of a view is more like the interface of a range or an iterator and views traditionally offer random access to elements in the view. The random access is practical in a parallel setting where the division of a range without random access can be complex.

6 My Benchmarking Library

This Chapter describes my library and the design choices that have were taken. Some implementation specific problems and solutions are also discussed here.

6.1 Design Goals

The goal of my library is to allow benchmarking of the parts of parallel library as well as parallel algorithms. This is not possible with any of the libraries discussed in Chapter 5 as it not possible to change the scheduler.

forcing the user to use a certain scheduler is not good library design. We have seen that a scheduler can improve the cache performance of for certain architectures. The choice of scheduler depends on the architecture on which the system runs. Furthermore scheduling is still a topic of active research and new schedulers need to be compared to existing schedulers. This requires a library where the performance effects of schedulers can be compared in a fair manner. Making it possible for the library to work with user defined schedulers support both the research in more effective schedulers and allows the library to be used effectively on many different architectures.

We have observed that nested parallelism makes both writing and reading parallel programs easier (see Section 4). The library should therefore support use of nested parallelism and make it easy for the user to program using the supplied methods of nesting parallelism.

Granularity is very important in practice (see e.g. [9]) and the performance of a parallel algorithm depends on the granularity. Granularity control is important, as optimal task granularity is depended on many parameters (see Section 5.1). Library support for granularity control makes experimentation easy and allow the user to easily change the granularity if changes are made to either software or hardware.

As seen in Section 4 division of data is a highly occurring operation in parallel algorithms. It should thus be made easy by the library. Working with Sections of data sequences are not trivial and can require a certain amount of boiler code. The library should try to minimize the amount of boiler code the user is required to write.

The library should avoid the use of a precompiler, to make bug finding easier. The library must thus be implemented in pure C++.

6.2 The Multi-core Library

Two main issues in the design is: the design of how tasks are spawned and the how a task is represented. These choice are interdependent and the design choices of one limits the available choices for the other. The design of the task representation and the spawn functionality is discussed first. The design rest the rest of the library is then described.

My library consists of three major components: a thread pool, a task class, and a scheduler. The final design of each component is shown on Figure 6.1.

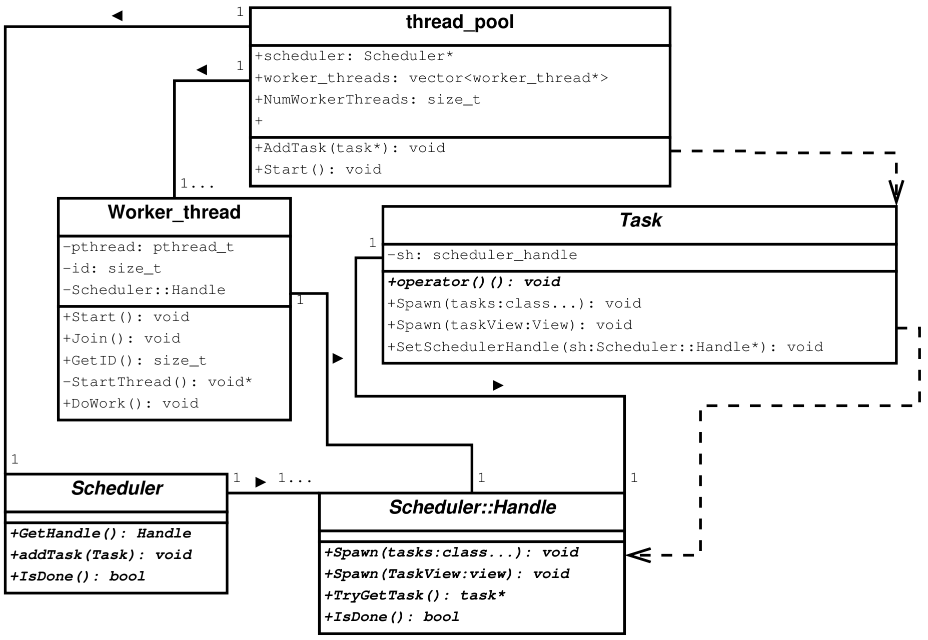


Figure 6.1: UML diagram of the core classes in the framework.

6.2.1 Tasks and Spawning

Because we want a pure C++system, the spawn functionality must be implemented as a C++function and the code to be run in parallel must thus be passed as arguments. This could be done with function pointers, but C++is a strongly typed language and the signature of a function depends on the types and number of arguments the function takes. The solution to this problem is to use function objects, called *functors*. A functor is an object that can also act as a function. This is achieved by declaring a method that is used to execute the functor (by convention `operator()` is used). To make the functor class declared by the user available to the system an abstract class `Task` is provided that the user-declared functors must inherit from.

Letting the spawn statement return a value is practical, but cannot be implemented for the same reasons as above, and the function that executes a `Task` functor must be declared as `void`. Returning can be achieved by explicitly writing the value to be returned at a given memory address. The `Task` functor thus take the addresses, to where the returned values should be written, as an argument. This allows spawn to return values indirectly as well as removing the requirement that only a single value be returned.

Because the spawn statement only takes a functor as argument and does not return anything there is no reason to limit spawn to only one task. The spawn statement is thus expanded to allow any number of tasks to be spawned. Traditionally the spawn statement is used like fork, splitting a task into two threads of execution. This results in a binary tree structure of spawn statements. The spawn tree allows the spawning to be done in logarithmic depth using the parallelism of the spawns. But, as discussed in Section 5.1, each task has an overhead from the scheduler and a task that is only used to spawn two new tasks, would have a very low granularity. The syntax of spawning several tasks as defined here allows each scheduler to implement the spawn in the most efficient way.

Because moving running code from one thread to another is both non-trivial and time expensive, a task must be executed from start to finish on the thread that starts executing that task. To make synchronization (like the Cilk `sync` statement) available each thread has a stack of running tasks. When a parent task 'syncs', the thread puts the parent task on the stack and runs other tasks until all children of the parent are completed. The execution of the parent is then resumed. This stack based synchronisation has two possible interfaces. The first interface has an explicit sync statement, as in Cilk. A sync is implicitly called when the task completes. The second interface makes the sync an implicit part of spawn. The spawn then becomes blocking and the parent task continues when all child tasks have completed. The two interfaces are equivalent in the functionality they can express, but differ in the natural way of thinking about the spawn statement. The first interface makes it natural to think in terms of threads of execution, each spawn splits the tread into several threads. The second interface makes it more natural to think in discrete tasks, where tasks can be made of more subtasks. It should be noted that the second interface results in one more task being spawned and thus a higher overhead. The second interface has a clearer separation of data between the spawned tasks. Any shared data is thus easier to find, making race-conditions and deadlocks easier to find.

The strategy for spawning described above, with blocking spawn using functors, has an additional benefit. It makes the functionality of Cilk hyperobjects easily available, by using data members in the functor a task can store a state. A spawn starts with the parent task constructing the tasks to be spawned. The tasks are then spawned and executed by the library. The execution of the parent task then continues, where the local values from the children tasks are combined in some way. The task functors are constructed by the parent task and it should be the responsibility of the parent task to deallocate any memory it has allocated. The child task functors and their fields are thus available to the parent task after the completion of the child tasks.

6.2.2 Thread Pool and Worker Threads

The thread pool is a standard way of overcoming the limit on the number of operating-system threads. The thread pool creates a number of worker threads.

Each worker thread runs the tasks specified by the scheduler.

The `WorkerThread` class is a wrapper around a platform dependent thread library, or a C++thread object. The `ThreadPool` creates the required number of `WorkerThreads` and initializes each with a scheduler handle. Each `WorkerThread` waits until it is started by the `ThreadPool`, before the platform thread is created and started. A worker thread loops until all tasks have been completed. At each iteration the worker thread requests a `Task` and then executes it. When all `Tasks` in the scheduler have completed the worker threads join with the main thread and the system exits.

6.2.3 Scheduling Design

The library should be able to allow use of user defined schedulers. The scheduler thus needs to be parameterized. To achieve this the basic interface of a scheduler needs to be identified. The interface should allow for as many different schedulers as possible. We use the two schedulers discussed in Section 3 as a basis for identifying this interface. The most prominent difference between the work-stealing scheduler and the parallel-depth-first scheduler is where the task queue is stored. The work-stealing scheduler stores a task deque for each thread. The parallel depth first scheduler stores a central task queue that all threads use concurrently. The scheduler interface should allow for both.

To support both kind of schedulers we need a central `Scheduler` object, for any shared data, and a per-thread object that can store any per-thread data. Thus we have a central `Scheduler` object that is hidden from the threads and a `Scheduler::Handle` class. Each `WorkerThread` holds an `Scheduler::Handle` that serves as the communication link between `WorkerThreads` and the central `Scheduler` object.

The `Scheduler::Handle` is used by the work stealing scheduler to allow the central scheduler to store a list of handles, that is used when a thread steals from a victim thread. To maintain the `Scheduler` and the `WorkerThread` decoupling the `WorkerThread` is given a `Scheduler::Handle` at creation by the `ThreadPool`. The thread pool creates the worker threads and requests a scheduler handle for each created `WorkerThread`.

The central `Scheduler` has is responsible for two operations. Adding a root `Task` to the schedule, i.e. the main `Task` of the program and a way to request a `Scheduler::Handle` object. A `WorkerThread` needs: a way to test if all `Tasks` have completed, a way to `Spawn` new `Tasks`, and a way to get the next `Task` that should be execute. These three operations are thus the responsibility of the `Scheduler::Handle`. This basic interface allows for implementation of both the work-stealing and the parallel-depth-first scheduler.

6.3 Implementational Details

Using a public task interface in the form of a minimal abstract task that the user-written functors can inherit from This would have been the preferable way to implement the task interface. It, however, proved to be too be too inflexible and impractical. The task is therefor implemented as part of the scheduler. User defined tasks can then be templated on the `Task` type that the scheduler use as a policy. The benefit of this policy based design is that the function can be specialized for a certain scheduler making it possible to optimize for a scheduler.

The `Spawn` function in the `Scheduler::Handle` first initialize any needed data members and then add the `Task` to the ready-pool data structure. After the `Task`s have been added to the ready-pool, the `Spawn` function saves the currently executing task on the stack and enters a loop. At each iteration a `Task` is fetched from the ready-pool and then executed. When all children of the parent task have completed the loop is stopped and execution of the parent task continues.

Inserting all spawned `Tasks` into the ready-pool directly works fine for any number tasks known at compile-time. To allow a dynamic number of tasks to be spawned the schedulers implement a spawn function that given a `View` of `tasks` spawns all tasks in the view. Spawning a dynamic number of `Task`s need extra care. If a very large number of `Tasks` are spawned performance could degrade, e.g. if the ready-pool cannot fit in L2 cache or if the ready pool and the input data cannot fit in main memory together. The two implemented schedulers handle the issue differently but they both use *lazy spawning* where the `Task` is only inserted in the ready-pool when it is needed.

The work stealing scheduler perform lazy spawning by recursive splitting of the view. the scheduler uses an internal `SplittingTask` class to represent a view Section. The `SplittingTask` is inserted into the ready pool as a normal `Task`. When a `SplittingTask` is executed it either splits the view into two subsections and spawn a `SplittingTask` for each Section or if the view contain less than a constant number of `Task`s they are inserted into the ready pool directly.

The parallel-depth-first-scheduler perform lazy spawning [4] by inserting a `ViewNode` into the task queue that represent the ready pool. A `ViewNode` contain a view of `Task`s and a iterator on the view of `Task`s. When a `Scheduler::Handle` asks the task queue for the next `Task`, the queue find the next node in the queue, if the node i a `ViewNode` the view node inserts the `Task` pointed to by the iterator into the queue, in front of the `ViewNode`, and moves the iterator forward by one. Then task in the newly inserted node is then returned to the `Scheduler::Handle`. When the iterator of a `ViewNode` reaches the end of the view of `Tasks` the `ViewNode` is removed from the `Task` queue.

7 Evaluation

In this Chapter we evaluate the performance of the work stealing and the parallel depth first schedulers. This comparison is done by executing the sorting algorithms described in Chapter 4. The performance evaluation is done by using the parallel programming library described in Chapter 6. This allows the same code to be executed using both the work stealing and the parallel depth first scheduler allowing a fair comparison of the two schedulers. In the rest of this Chapter each result is presented and discussed.

During the benchmarking it was necessary to make the algorithms more computationally heavy, this is accomplished by changing the comparison function used in the sorting algorithms from a direct comparison of values to a comparison of the natural logarithm of the values. The results from this change is very interesting and the benchmarks for both are presented here. The same comparison function is used for both algorithms and the sequential sorting algorithm. The sequential algorithm should make fewer comparisons than the parallel algorithms as neither of the parallel algorithms are work efficient and perform extra comparisons. Changing the comparison function can thus be used to make the algorithms more computationally heavy without changing the speedup of the parallel algorithms.

All the benchmarks programs were compiled using g++ 4.5.2 with level three optimizations and were run on a Intel core 2 Quad Q9300 CMP, with four cores each of which runs at 2.5GHz. The CPU is made of two dual core CMPs each of which have a L2 cache of 3MB and each processor have a L1 cache of 32 KB. The effect of not having a cache that is shared by all processors is discussed. The benchmark computer has 4GB of DDR3 1333 RAM running in dual channel mode at 666MHz. The computer was running Ubuntu 11.04 with the kernel version 2.6.38-11 during the benchmarking.

7.1 Performance of Multi-Way Mergesort

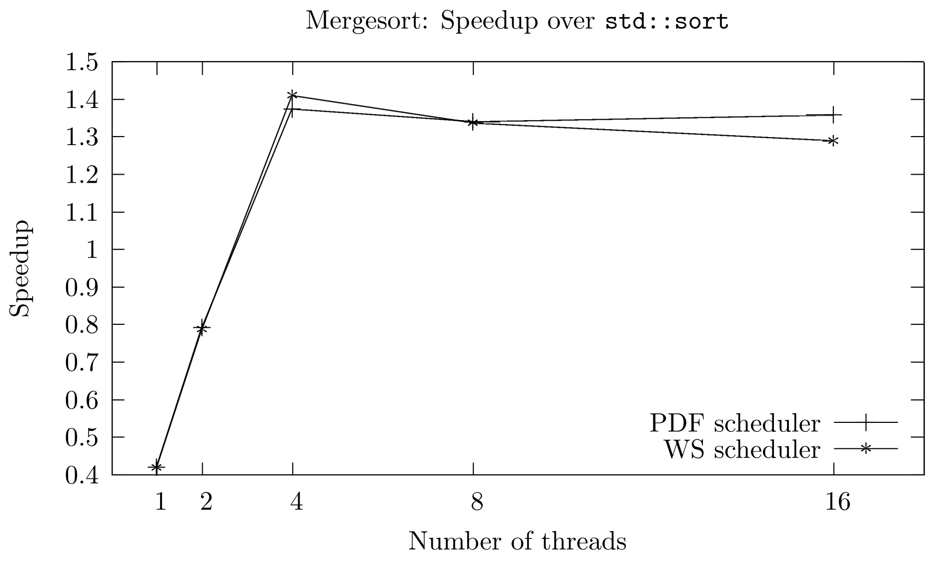
The mergesort algorithm was run on data with 8 byte elements. The input data was generated by initialising each position in the input array to its index and then shuffling the data. The input data is then divided into 16 runs and all runs were sorted in parallel using the sorting algorithm supplied by gcc. The runs were then merged using the parallel multi-way merging algorithm described in Section 4.1. The wall-clock time is measured for both the sorting of runs and the merging of runs individually. The timings are averaged over three executions.

7.1.1 Speedup of Multi-Way Mergesort

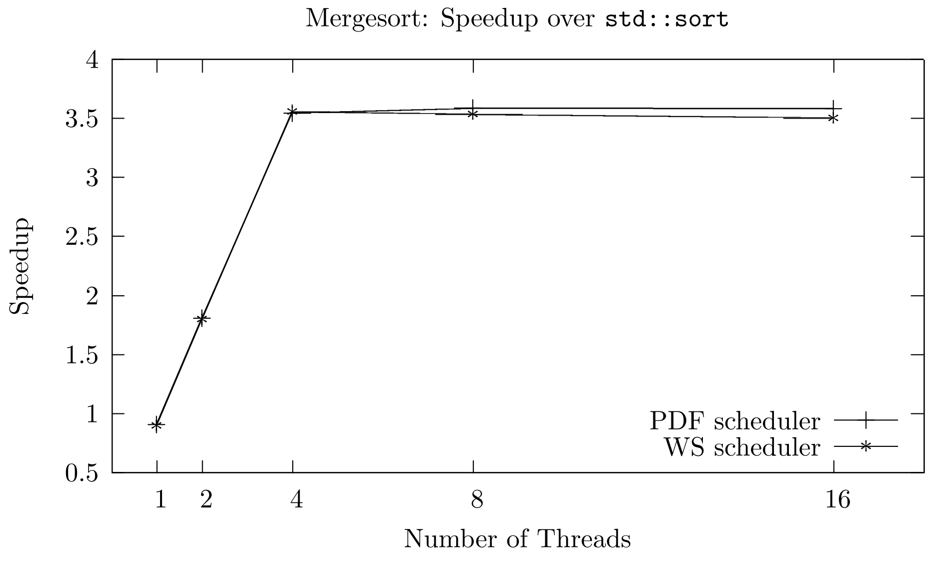
0.5 GB input data has been sorted using 16 runs, for both the direct value comparison and logarithmic comparison. The speedup is calculated using the sequential sorting algorithm from gcc and the total time used by the parallel multi-way mergesort algorithm. Figure 7.1 shows the speedup as a function of the number of threads used, with direct value comparison and logarithmic comparison. The speedup obtained when logarithmic comparison is used is more than double the speedup obtain with direct value comparison.

Figure 7.2 shows the scaling over the single-thread execution of the parallel mergesort algorithm with direct value comparison and the logarithm comparison. These figures do not show the formally defined speedup as it is defined is terms of best sequential algorithm, but they show how the parts of the algorithm scale. Because the overheads introduced by parallelization of the algorithm are present in the single-thread execution, the algorithm is expected to exhibit linear speedup. The multi-way merging algorithm scales to a bit less than the expected 4, with direct value comparison. The sorting of runs scales to 3, with the direct value comparison, which is less than expected. Both the merging and the sorting of runs scales as expected when the logarithm comparison is used.

The limited scaling for the run sorting is not expected as no synchronization is performed. That the run sorting scales just as expected when logarithmic comparison is used, leads us to conclude the off-chip bandwidth is the limiting factor. The four processors can sort the data faster than it can be fetched from main memory.

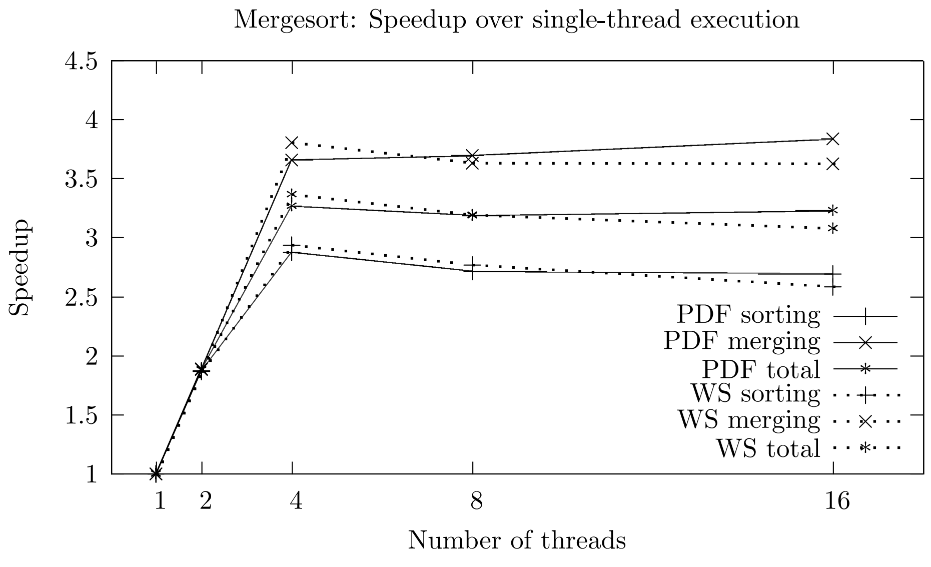


(a)

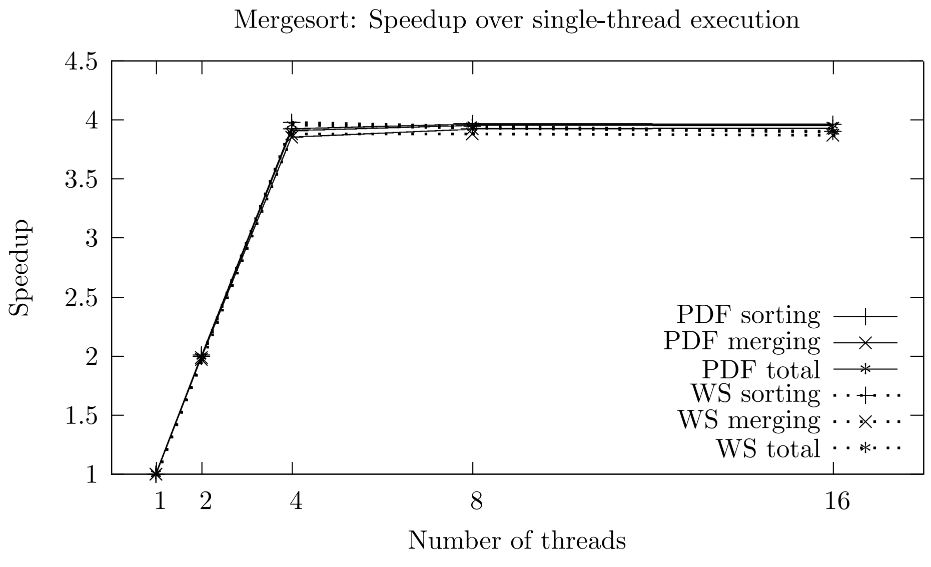


(b)

Figure 7.1: The speedup of the parallel multiway mergesort from the sequential sorting algorithm that comes with gcc. (a) shows the speedup with direct value comparison. (b) shows speedup with logarithmic comparison. Both were run with 0.5 GB input data using 16 runs.



(a)



(b)

Figure 7.2: The scaling of the parallel multiway mergesort shown is terms of speedup over the sequential execution of the parallel algorithm. (a) shows the scaling then direct value comparison is used. (b) shows the scaling when logarithmic comparison is use. Both were run with 0.5 GB of input data using 16 runs.

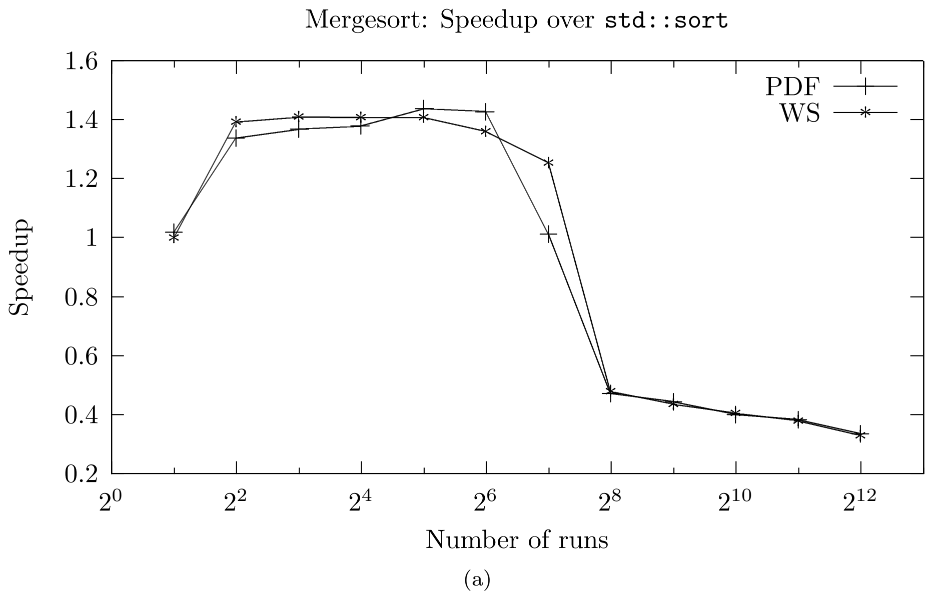
7.1.2 Granularity Benchmark of Multi-Way Mergesort

The granularity of the parallel multi-way mergesort is defined by the number of runs used. The effect of the granularity is measured by running the algorithm with an increasing number of runs while holding the total input size constant.The amount of input data varies between the direct value comparison and the logarithmic comparison benchmarks, which is 0.5 GB and 0.25 GB respectively. Due to very large change in execution time, the same amount of data was not used. The input data size was instead chosen to keep the execution time high enough, to make uncertainty of time measurements non-significant, and low enough that the benchmarks could be done in no more than a few hours. The results for the benchmarks can, despite the different input size, still be compared and the difference only results in an extra measure point in the direct value comparison plots. All four granularity benchmarks were performed using four threads.

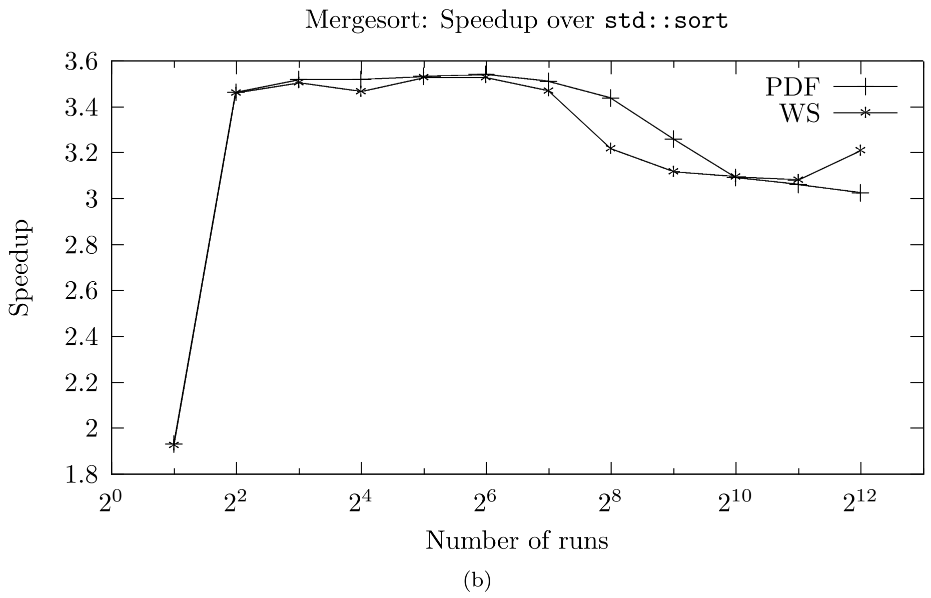
Figure 7.3 shows the speedup of the multi-way mergesort algorithm over the sequential sorting algorithm form gcc as a function of the number of runs used. The speedup of the direct value comparison is much lower than the logarithmic comparison is, as explained in Section 7.1.1, due to off-chip bandwidth. Both benchmarks show a drop in speedup between 64 and 256 runs.

Figure 7.4 shows the execution time of the multiway-merge sort algorithm over single-thread execution as a function of the number of runs, for direct value comparison and logarithmic comparison. The run sorting time is expected to decrease slightly as the number of runs increase, due to the log-linear complexity of sorting. The time spent on multi-way merging is expected to increase with increasing number of runs. Both run sorting and merging behaves as expected when logarithmic comparison is used. The high run sorting time for the first measure point is due to only two runs being used and only two processors can be used for sorting. The multi-way merging can used all four processors independent of the number of runs. The time spend on multi-way merging with direct value comparison show a steep increase between 64 and 256 runs.

The dramatic increased time used for merging and the corresponding decrease in speedup between 64 and 256 runs is probably due to the heap used for multi-way merging being too large to fit in L2 cache. This causes the contention on off-chip bandwidth to increase dramatically resulting in the observed drop in performance. This also fits with the logarithmic comparison benchmarks that drops slightly at the same point, seen on Figure 7.3b.

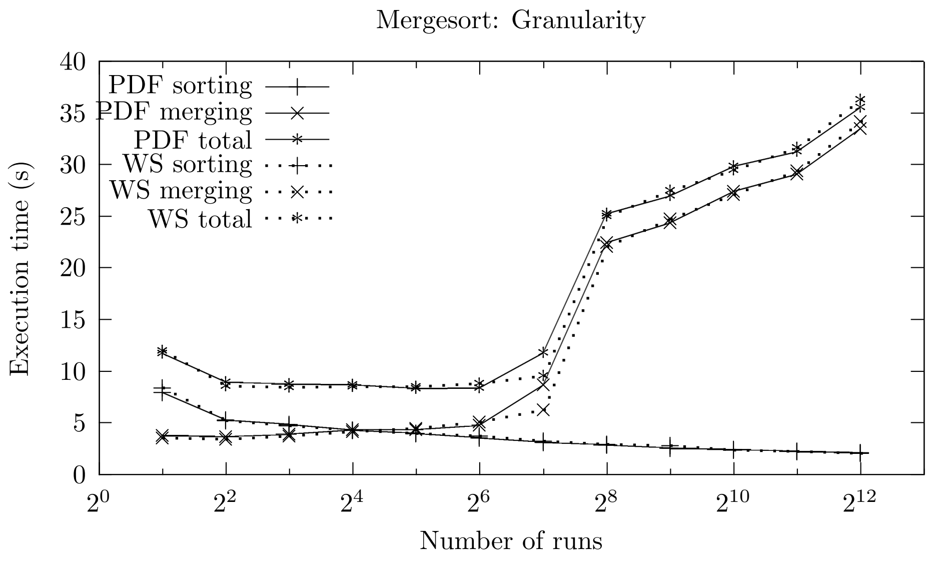


(a)

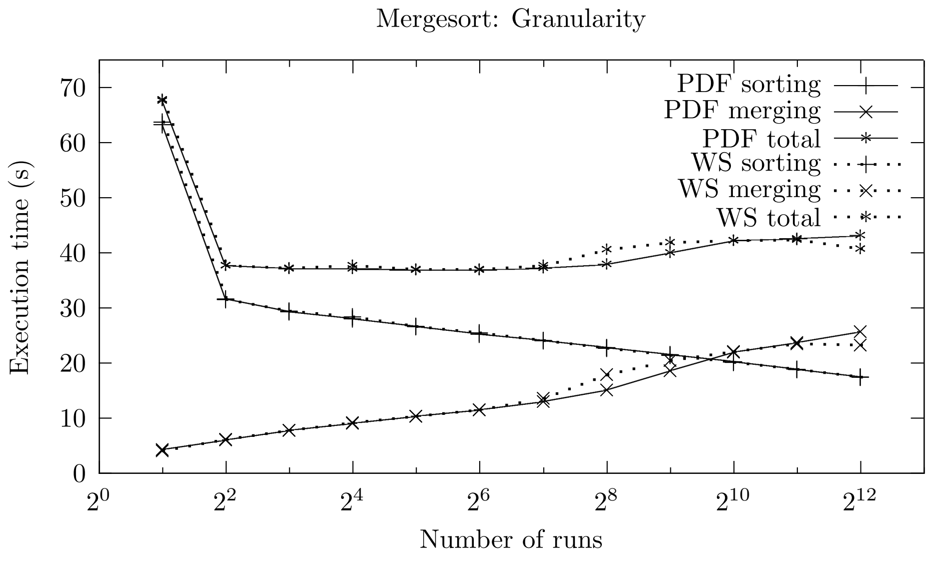


(b)

Figure 7.3: The effects on speedup of changing the number of runs is shown. (a) shows the speedup when direct value comparison is used. (b) shows the speedup when logarithmic comparison is used. 0.5 GB input data was used for direct value comparison ((a)) and 0.25 GB data was used for logarithmic comparison ((b)), both used 16 runs.



(a)



(b)

Figure 7.4: The effects of the number of runs used on the scaling of the run sorting and the merging, in terms of speedup over single-thread execution. (a) shows the scaling with direct value comparison. (b) shows the scaling with logarithmic comparison. 0.5 GB input data was used for direct value comparison ((a)) and 0.25 GB data was used for logarithmic comparison ((b)), both used 16 runs.

7.2 Performance of Parallel Quicksort

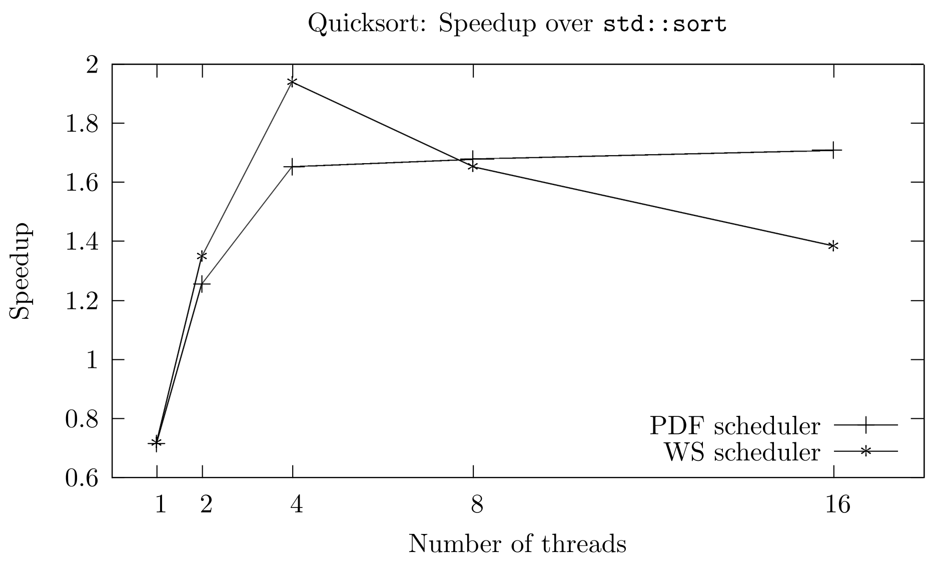
The quicksort algorithm was run with input data with 8 byte elements. For the direct value comparison benchmarks an input size of 0.5 GB was used and for the logarithmic comparison an input size of 0.25 GB was used. As in the mergesort benchmarking the input was generated by initialising each position in the array to its index and then shuffling the data.

7.2.1 Speedup of Parallel Quicksort

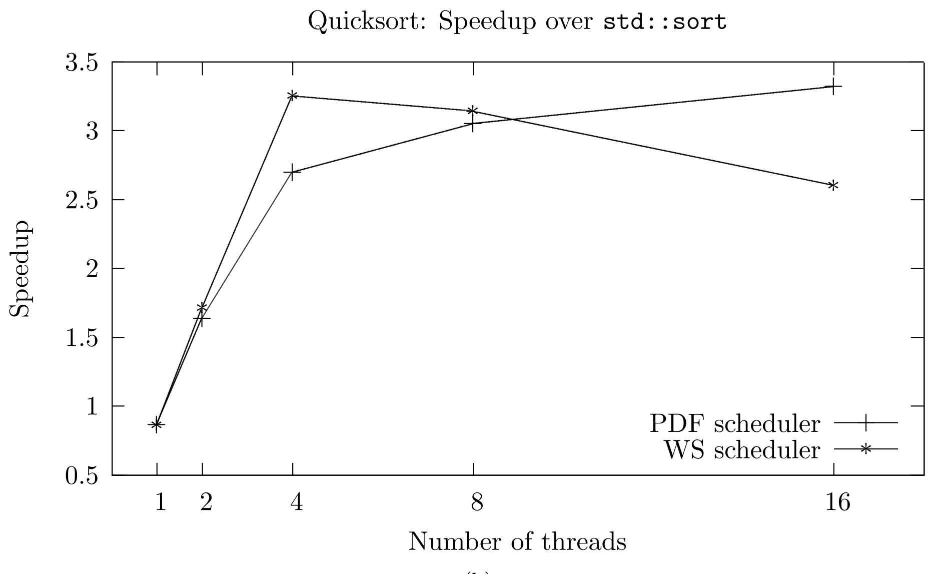
The speedup of the quicksort was measured by sorting either 0.5 GB or 0.25 GB of input data and measuring the wall-clock time used. The algorithm was run using 524288 elements as the direct sort size and a block size of 65536 elements, the number of tasks used for partitioning was limited to 16. Figure 7.5 shows the speedup of the quicksort algorithm over the sequential sorting algorithm from gcc, using direct value comparison and logarithmic comparison. Figure 7.6 shows the scaling of the quicksort algorithm, as the speedup over single-thread execution, using direct value comparison and logarithmic comparison.

The speedup and scaling of the quicksort algorithm are unsurprisingly very similar. We can see that the algorithm cannot use all four processors with direct value comparison. The algorithm scales to the expected four when logarithm comparison is used. This indicates that off-chip bandwidth is the limiting factor. The quicksort algorithm does not achieve as good speedup as the mergesort algorithm, even with logarithmic comparison.

Interestingly, a difference in the performance of the two schedulers, can been seen. The work-stealing scheduler scales to nearly 4 with four threads, using logarithmic comparison, whereas the parallel-depth-first scheduler only scales to 3 with four threads, also using logarithmic comparison. This likely indicate that work-stealing better use the limited bandwidth for this specific algorithm, but this should be investigated further before any conclusions can be made. Another difference in the performance of the two schedulers is their behaviour when more threads are used, than there are processors available. The performance of the quicksort algorithm degrades when more than four threads are used, under the work-stealing scheduler. This is most likely caused by increasing number of context switches. The work-stealing scheduler is unable to use the additional threads and adding additional threads forces the processor to use time slicing resulting in an increasing number of context switches. The quicksort algorithm performs better when more than four threads are used with the parallel-depth-first scheduler. The algorithm does not scale to a speedup of four over the single-thread execution, for four thread using the PDF-scheduler. This means that the processors are not used optimally and adding more threads per processor allows the processor to have a bit more available work resulting in faster execution.

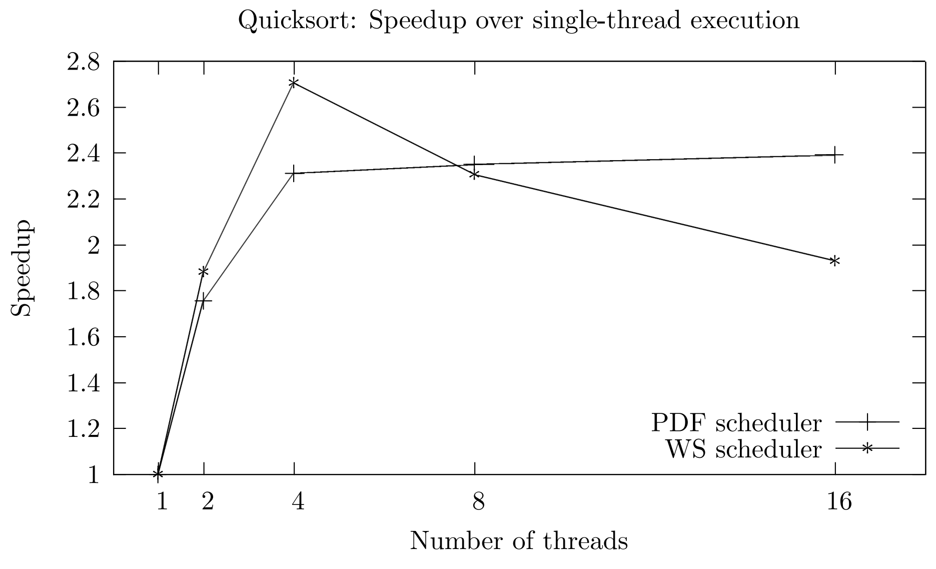


(a)

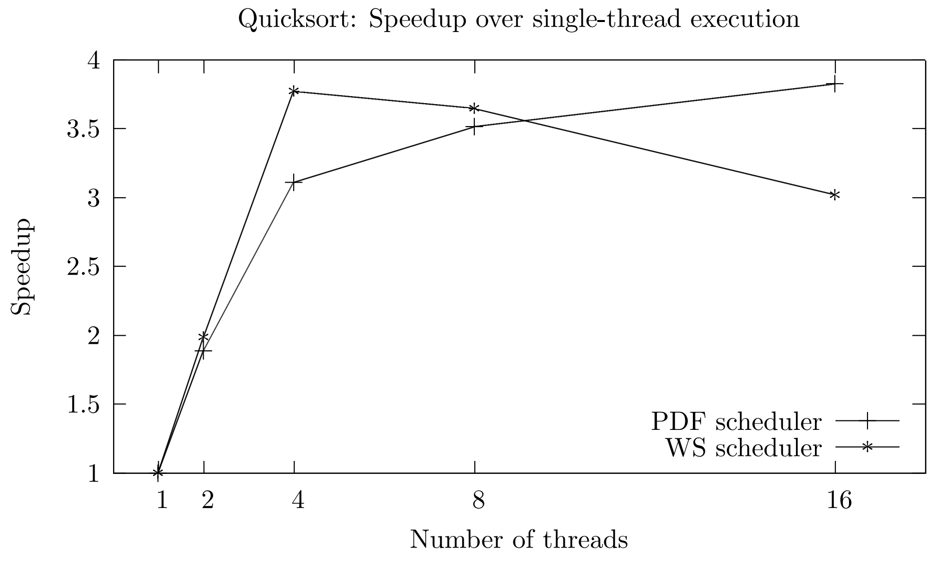


(b)

Figure 7.5: The speedup of the parallel quicksort algorithm compared to the sequential sorting algorithm from gcc. (a) shows the speedup with direct value comparison. (b) shows the speedup with logarithmic comparison. The direct value comparison benchmark ((a)) was run with 0.5 GB of data. The direct value comparison benchmark ((b)) was run with 0.25 GB of data. Both benchmarks were run with direct sorting of 2^{19} elements and a block size of 2^{16}.



(a)



(b)

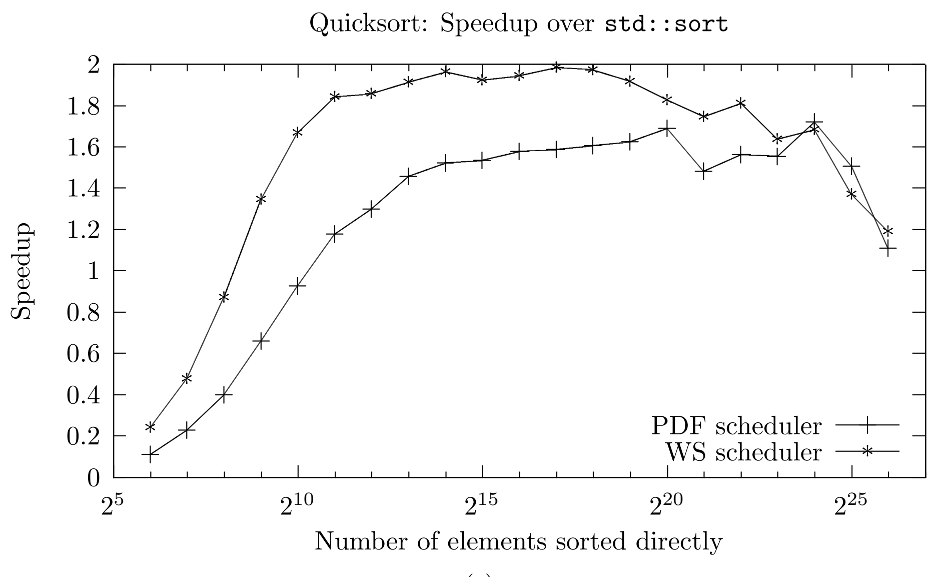
Figure 7.6: Scaling of the parallel quicksort algorithm, in terms of speedup over single-thread execution. (a) shows the scaling with direct value comparison. (b) shows the scaling with logarithmic comparison. The direct value comparison benchmark ((a)) was run with 0.5 GB of data. The direct value comparison benchmark ((b)) was run with 0.25 GB of data. Both benchmarks were run with direct sorting of 2^{19} elements and a block size of 2^{16}.

7.2.2 Granularity of Parallel Quicksort

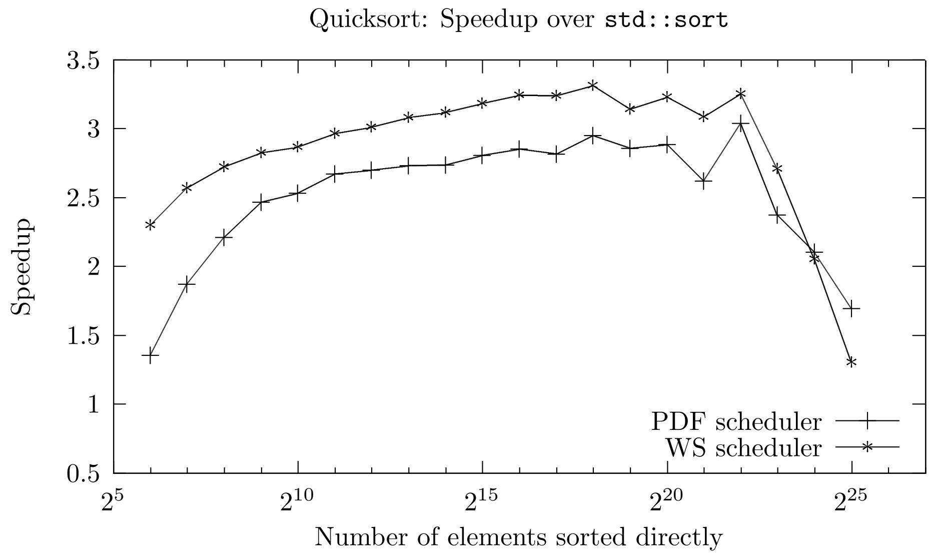
The task granularity effects on performance was measured. The task granularity was controlled by changing the number of elements that were sorted directly. The block size used during partitioning was set to one fourth of the number of elements sorted directly. The number of tasks was limited to four and four threads were used. Figure 7.7 shows the speedup of the quicksort algorithm using direct value comparison and logarithmic comparison. Figure 7.8 shows the execution time as a function of the task number of elements sorted directly, for both direct value comparison and logarithm comparison.

The quicksort algorithm behaves as expected when logarithmic comparison is used. The performance with very low granularity is dominated by the scheduler overhead. With very high task granularity the available parallelism is limited, decreasing the performance. The performance does not decrease by much for large task granularity, when direct value comparison is used. This is because the data is partitioned at least once, and the performance gained by four threads instead of two marginal (see Figure 7.6a).

The quicksort algorithm is generally faster with work-stealing scheduler. At low task granularity this is likely due to the contention on the central task queue used by the PDF scheduler. At larger task granularity this corresponds to our the observations on the speedup discussed in Section 7.2.1

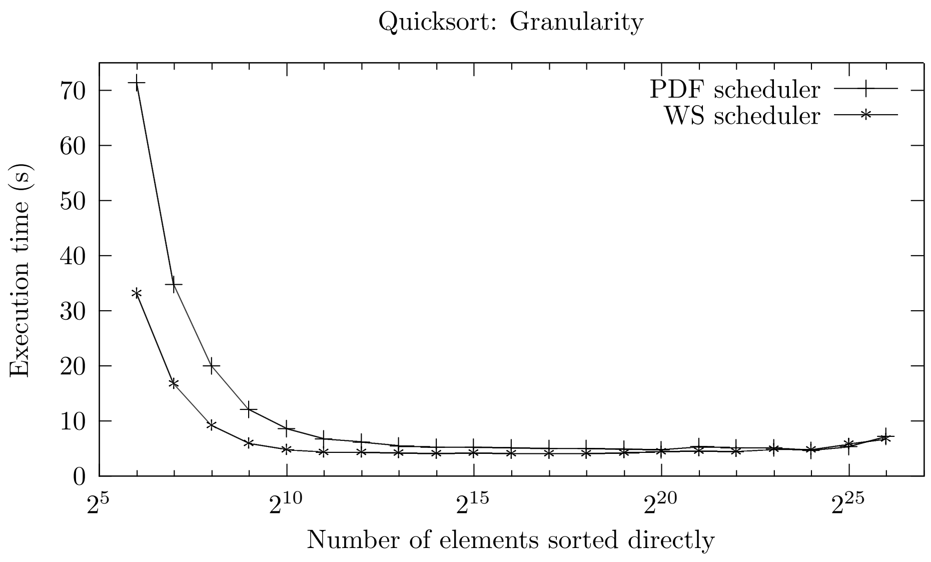


(a)

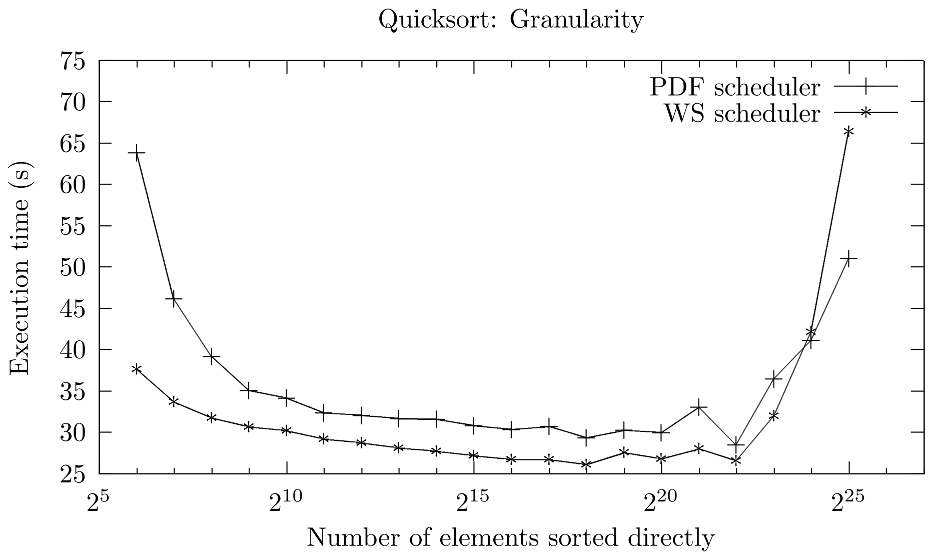


(b)

Figure 7.7: The effect of task granularity on speedup. (a) shows the speedup when direct value comparison is used. (b) shows the speedup when logarithm comparison is used. The direct value comparison benchmark ((a)) was run with 0.5 GB of data. The direct value comparison benchmark ((b)) was run with 0.25 GB of data. Both use a block size of a fourth of the number of elements sorted directly and a limit of four tasks for partitioning.



(a)



(b)

Figure 7.8: The effect of task granularity on the execution time of the quicksort algorithm. (a) shows the execution time with direct value comparison. (b) shows the execution time with logarithmic comparison. The direct value comparison benchmark ((a)) was run with 0.5 GB of data. The direct value comparison benchmark ((b)) was run with 0.25 GB of data. Both use a block size of a fourth of the number of elements sorted directly and a limit of four tasks for partitioning.

8 Conclusions

Chen et al. [9] compared the performance of the parallel-depth-first scheduler to work-stealing. They used mergesort, hash join and the LU benchmark for the comparison. Their mergesort algorithm used a parallel binary merging procedure. They used simulated execution of the algorithms to obtain results for CMPs with many more cores than are available today. Their results show a difference that becomes noticeable at eight processors for mergesort and four for hash join. The LU benchmarks showed no difference in scheduler performance. They furthermore discovered that the parallel-depth-first scheduler is more sensitive to the task granularity, but when the granularity is tuned mergesort ran 1.17 times faster with parallel depth first scheduling than with work stealing when 32 core was used.

The mergesort used by Chen et al. [9] and the quicksort used in this work are of similar structure, as both algorithms use a parallel binary partition or merge procedure and a sequential algorithm as the basis for solving sufficiently small tasks. The results for the two should thus yield comparable results. This is however not the case; the results for the quick sort obtained here have much lower scalability than the results shown in [9]. The scalability and speed up for both algorithms and both schedulers are not as high as expected. Only the merging procedure show a scalability as expected.

The simulation results Chen et al. present do not show the same bandwidth bottleneck that we observe. Chen et al. do not discuss the off-chip bandwidth at all and it is not mentioned if this is part of the simulation.

Our benchmarking were performed on a CMP without a cache shared by all processors. The CMP used thus does not fit with the CMP-cache model. This do not make the benchmarking of the PDF scheduler invalid. The PDF scheduler should still make better use of the cache that the processor pairs share. We use a single CMP and even though the fully shared cache is not present is both L2 caches on the same chip and fetching a cache line from the other L2 cache is still much faster than fetching it from main memory. Adding a shard cache will not increase the memory bandwidth, which is the current bottleneck. Only when the algorithm is made more computationally heavy is it possible to use the full CMP.

It is unlikely that even if the algorithms were not limited by off-chip bandwidth that we would be able to observe any large differences in the performance of the two schedulers, unless many more processors were available. We can however observe that the choice of algorithm is important and the parallel multi-way mergesort algorithm generally out preform the parallel quicksort algorithm due to the better memory bandwidth utilization.

The benchmarks of the parallel algorithms show us a important problem with CMPs; the limited off-chip bandwidth. It is a big problem that the processor is so fast and the main memory so slow that nearly no speedup is possible for sorting. Future CMPs are expected to have even more processors but if the off-chip bandwidth is not increased significantly using the added processors will not be possible.

The main memory that is used with modern CMPs have not changed much from the memory used in previous single processor computers, the most significant change is a slow increase in access speed. This makes the multiple processors in a CMP hard to utilize as the memory access is mostly sequential.

The difference in the performance of the parallel multi-way mergesort and the parallel quicksort algorithms demonstrate that algorithm design can help limit the off-chip bandwidth bottleneck. This a topic that requires more research, as it is not possible to use the results from the external-memory model and the ideal-cache model because of the added parallelism.

The cache models in processors today are very complex and subtle effect are present. This have not been a problem when only a single processor is used, where the cache could be modelled in general allowing effective cache utilization on most processors. This general model is not available for current CMPs as they are too different. Hardware manufactures are increasing the cache sizes in attempt to increase performance, while experimenting with the architecture. This make effective programming for general CMPs hard.

Bibliography

[1] A. Aggarwal and S. Vitter, The input/output complexity of sorting and related problems, *Communications of the ACM* **31** (1988), 1116–1127.

[2] G. Blelloch, R. Chowdhury, P. Gibbons, V. Ramachandran, S. Chen, and M. Kozuch, Provably good multicore cache performance for divide-and-conquer algorithms, *Proceedings of the 19th annual ACM-SIAM Symposium on Discrete Algorithms*, ACM/SIAM (2008), 501–510.

[3] G. Blelloch and P. Gibbons, Effectively sharing a cache among threads, *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures*, ACM (2004), 235–244.

[4] G. E. Blelloch, P. B. Gibbons, and Y. Matias, Provably efficient scheduling for languages with fine-grained parallelism, *Journal of the ACM* **46** (1999), 281–321.

[5] R. Blumofe and C. Leiserson, Scheduling multithreaded computations by work stealing, *Procedings of the 35th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society (1994), 356–368.

[6] A. Buss, A. Fidel, H. Harshvardhan, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. Amato, and L. Rauchwerger, The STAPL pview, *Languages and Compilers for Parallel Computing*, *Lecture Notes in Computer Science*, Springer-Verlag (2011), 261–275.

[7] A. Buss, H. Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger, STAPL: standard template adaptive parallel library, *Proceedings of the 3rd Annual Haifa Experimental Systems Conference*, ACM (2010).

[8] J. Casazza, *First the Tick, Now the Tock*, Intel Corporation (2009).

[9] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch, B. Falsafi, L. Fix, N. Hardavellas, T. C. Mowry, and C. Wilkerson, Scheduling threads for constructive cache sharing on CMPs, *Proceedings of the 19th Annual ACM Symposium on Parallel Algorithms and Architectures*, ACM (2007), 105–115.

[10] R. Cole and V. Ramachandran, Resource oblivious sorting on multicores, *Procedings of the 37th International Colloquium on Automata, Languages and Programming*, *Lecture Notes in Computer Science*, Springer-Verlag (2010), 226–237.

[11] T. H. Cormen, C. E. Leiserson, R. L. Riverst, and C. Stein, *Introduction to Algorithms*, 3th Edition, The MIT Press (2009) Chapter 18.

[12] E. D. Demaine, Cache-oblivious algorithms and data structures, Worldwide Web Document (2002). Available at `http://erikdemaine.org/papers/BRICS2002/`.

[13] U. Drepper, What every programmer should know about memory, Worldwide Web Document (2007). Available at `http://people.redhat.com/drepper/cpumemory.pdf`.

[14] M. Frigo, P. Halpern, C. E. Leiserson, and S. Lewin-Berlin, Reducers and other Cilk++ hyperobjects, *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, ACM (2009), 79–90.

[15] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, Cache-oblivious algorithms, *Proceedings of the 40th Annual IEEE Symposium on Foundations of Computer Science*, IEEE Computer Society (1999), 285–298.

[16] M. Frigo, C. E. Leiserson, and K. H. Randall, The implementation of the Cilk-5 multithreaded language, *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ACM (1998), 212–223.

[17] Intel, Threading building blocks reference manual, Worldwide Web Document (2011). Available at `http://threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference.pdf`.

[18] J. Laudon and D. Lenoski, The SGI origin: a ccNUMA highly scalable server, *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ACM (1997), 241–251.

[19] K. Mehlhorn and P. Sanders, *Data Structures and Algorithms*, Springer verlag (2008) Chapter 2.

[20] J. Singler and B. Konsik, The GNU libstdc++ parallel mode: software engineering considerations, *Proceedings of the 1st International Workshop on Multicore Software Engineering*, ACM (2008), 15–22.

[21] J. Singler, P. Sanders, and F. Putze, MCSTL: The multi-core standard template library, *Procedings of the 13th International Euro-Par Conference*, Springer verlag (2007), 682–694.

[22] P. Tsigas and Y. Zhang, A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000, *Procedings of Euromicro Conference on Parallel, Distributed, and Network-Based Processing*, IEEE Computer Society (2003), 372–391.

[23] P. J. Varman, S. D. Scheufler, B. R. Iyer, and G. R. Ricard, Merging multiple lists on hierarchical-memory multiprocessors, *Journal of Parallel and Distributed Computing* **12**, 2 (1991), 171–177.

[24] J. S. Vitter, Algorithms and data structures for external memory, *Foundations and Trends in Theoretical Computer Science* (2006), 305–474.

[25] M. Weiser and G. Powell, The view template library, *Proceedings of the 1st Workshop on C++ Template Programming* (2000).