Comparison of OpenMP and Threading Building Blocks for expressing parallelism on shared-memory systems

Peder Rindal Refsnes

Institutt for informatikk
Universitetet i Bergen

Master thesis
2011

Contents

1 Introduction

There is currently a shift underway in processor architecture. Previously, one could expect the performance of a program to increase as faster and faster processors were introduced. Eventually, processor manufacturers reached the limit of what was physically viable. Increasing clock frequencies and smaller die sizes create more and more heat, so much so that it became impractical to cool them by conventional means. *Moore's law*, somewhat paraphrased, states that the number of transistors doubles every eighteen months. Single core processors are no longer able to convert these added transistors to increased performance. Instead, manufacturers have sought to keep Moore's law relevant by adding several central processing units (CPUs) on a single die, dubbed *multi-core* or *many-core* processors. These rely on several simpler cores which communicate through a system bus or shared hardware caches. By offering processors with more and more cores one can expect the performance of programs to keep increasing. Unlike single core processors where increased speed directly translates to improved performance, this improvement does not come for free.

Parallel computing is a field of study where problems are broken up into smaller ones and solved in parallel. The area has seen much research going back as far as the fifties. This was first a purely theoretical field, but later put into practice for solving computational intensive problems, dubbed *high performance computing*. Also known as supercomputers, these systems can contain several thousand processing units connected in a cluster. Each node in a cluster has its own memory to store data and send messages to each other when the need for communication arises. This approach is called *message passing*. Message passing scales well for problems where each node can mostly work independently of the rest of the cluster. This is because the major

bottleneck is the communication channel. This form of parallelism is referred to as *coarse grained parallelism*.

Multi-core systems, as mentioned, communicate through shared-memory and can access the same data structure instances. Such systems are referred to as *shared memory systems*. An advantage when compared to message passing is lower cost of communication in terms of latency. This allows for a much more fine-grained levels of parallelism to be achieved efficiently. A usual strategy when using large computer clusters is to split a large problem into reasonably independent parts and distribute each part to a node. Each node is a shared memory system. Supercomputers are programmed mostly by researches or other experts in the field of parallel computing. Previously these were programmed using raw threads. This means to work directly with the thread API provided by the operating system. Using raw API threads, like POSIX threads (pthreads) on UNIX systems or Windows threads, is informally referred to as the assembly language of parallelism. Like assembly programming, raw threads provide a great deal of flexibility at the expense of a high cost in effort and complexity. Another common approach was for each computer vendor to provide their own set of *parallel directives*. These allowed a programmer to express parallelism by communicating to the compiler which constructs, typically loops, should be run in parallel. The syntax of the parallel directives varied from system to system since no standard was in place.

With the introduction of multi-core processors in commodity computers and mobile phones, parallel computing is no longer considered a niche in programming, but something every programmer has to be aware of. It is not expected that every programmer should become an expert in parallel programming. A lot of research has therefore been done and approaches have been introduced to abstract away the hardware details. These approaches have been realized in programming languages specifically designed to express parallelism and various libraries. Working directly with threads is complex, time consuming and error prone. By building our parallel programs with abstractions we can get more done, in less time and with fewer bugs.

The focus of this thesis is approaches that allow the programmer to express parallelism in current programming languages. More specifically, using OpenMP [4] and Intel Threading Building Blocks (TBB) [18] to express parallelism in C++. These approaches provide abstractions to relieve programmers from having to work directly with threads. In the remainder of this chapter we will give background information about parallel programming. This includes some fundamental theory about maximum expected improvement, the terminology used throughout this thesis as well as brief sections about the programming language used in the implementations and physical processor considerations.

In Chapter 2 we formally present OpenMP and TBB. A brief history of each approach is given. Following this is a description of the terminology used in the literature of each approach as well as a trivial example to give the reader a feel for the syntax. The chapter concludes with a comparison where the strengths and

weaknesses of each approach is discussed.

In Chapter 3 several implementations of the mergesort sorting algorithm are presented. This serves as an thorough example of how to achieve speedup with the two approaches. Algorithms using both iterative and recursive techniques are presented, analyzed and implemented. The implementations are then tested and results presented showing favorable performance when compared to the parallel sorting algorithm provided by the TBB library.

Chapter 4 presents the first non-trivial parallel program and deals with finding connected components in graphs using disjoint set data structures. Several variations of union-find algorithms are presented and implemented with OpenMP and TBB. In addition to implement algorithms based on mutual-exclusion, a novel technique which make use of an additional verification step is presented. An implementation of Rems algorithm is shown to scale well in all tests performed. This suggests that the parallel algorithm presented could have practical implications.

Chapter 5 sums up the thesis by suggesting further work and some closing thoughts on OpenMP and TBB.

1.1 Maximum expected improvement

Amdahl's law

The *speedup* S of a program is defined as the reduction in running time when adding more processing units. In this section we will look at what speedups we can expect to get. Ideally, doubling the number of processor from p to $2p$ would lead to a halving in the time it takes to compute a solution. This is sadly, often not the case. Many problems have some parts that provably needs to be executed sequentially. This fact was covered in [1] which would later be referred to as *Amdahl's law*. Amdahl's law describes the theoretical maximum speedup we can expect when a computation is carried out using p processors, assuming solving the problem sequentially gives $S = 1.0$. Let f be the fraction of the problem that can be run in parallel. The sequential part then takes time $1 - f$ and the parallel part takes time $\frac{f}{p}$. Overall the computation takes time $1 - f + \frac{f}{p}$. Amdahl's law says that the best speedup we can hope for is $S = \frac{1}{1-f+\frac{f}{p}}$ What does this say about the speedup we can achieve? Let us assume we have a problem where 10% needs to be run sequentially, and the remaining 90% can be run concurrently. This gives us $f = \frac{9}{10}$. If we had a processor with 16 cores the best speedup we could hope for is $S = \frac{1}{\frac{1}{10}+\frac{9}{16}} = 6.4$. As the number of cores increase toward infinity a larger and larger percent of the time is spent in the sequential part. This is a long way from the $S = 16$ we can hope to achieve when dealing with an *embarrassingly parallel* problem. Embarrassingly parallel problems are defined as problems where $f = 1$. In other words, problems where no communication between processors is necessary.

Gustafson's law

The situation seems pretty dire when the repercussions of Amdahl's law are considered. Some years after Amdahl published his famous paper another researcher presented a more optimistic view. This view is known as *Gustafson's law*. In [17], Gustafson points out a wrong assumption made by Amdahl, the size of a problem instance is not fixed. What we are really trying to do is compute the solution for the largest problem size possible in some "reasonable" time. More formally the law reads $S_{p,n} = a(n) + p * (1 - a(n))$, where S refers to the speedup, p to the number of processors, n the problem size and a is the sequential part of the problem. Assuming that $a(n)$ decreases with increasing values of n, the speedup S approaches p as n approaches infinity. In other words, as we add more and more processors the fraction of time spent in the sequential part decreases. This law redefined the meaning of efficiency to mean reducing the sequential part of a program, even if it increases the total amount of computation.

Parallel paradigms

Programming multi-core systems add some extra issues the programmer needs to be aware of. Because of the added complexity involved when sharing data between processors, one must make sure that data is accessed in a safe manner. The lack of such guarantees may lead to indeterminate program behavior and hard-to-find bugs.

When reasoning about multicore systems we imagine multiple threads running in parallel performing operations on some shared objects. It is trivial to imagine some scenario where several threads attempt to update an object simultaneously and leave the object in a non-correct state. Thus, there needs to be some mechanisms for ensuring correct execution.

There are several semantics that all approaches to parallelism need to support, implicitly or explicitly. These include, in no particular order, constructs for *work-sharing*, *load balancing*, *mutual exclusion* and *synchronization*. Work sharing refers to how we distribute the problem set over available threads. Splitting up a problem can be a non-trivial task since equal ranges do not necessarily imply equal work. As an example, suppose we are trying to find all prime numbers from 1 to n. A naive approach would split the input in n/p ranges where p is the number of concurrent threads. There are several problems with this approach. First, prime numbers are not evenly distributed among the set of integers. Secondly, it takes longer to check whether a value close to n is prime or not. This example is contrived, a better way to find primes would make use of the Sieve of Eratosthenes, but it shows the need for a way to specify how the work should be shared. Load balancing refers to the distribution of work over several threads. Mutual exclusion constructs ensures that accesses to shared objects is limited to one thread at a time. This can be realized through the use of locks, or through transactions. Synchronization constructs allows

the programmer to declare which parts of an algorithm has to be completed by all running threads before execution can commence.

We usually distinguish between two different forms of parallelism; data and task parallelism. Data parallelism means applying the same operation on each element of a data set. An example of this would be applying a squaring function on each element of an array of integers. An example of special hardware that excels at data parallelism are graphic processing units (GPU) which offer an order of magnitude more concurrent threads than are available on CPUs, but with several limitations. For instance, on a typical GPU, eight or more processors share a common instruction pointer. Data parallelism is not the focus of this thesis but is mentioned for completeness.

The other, task parallelism, allows entirely different parallel operations to be performed on each element. In task parallelism a problem is broken down into distinct task. These may or may not need to be performed in a given sequence. Note that it is possible to have a problem consisting of only one logical task. In this case each task instance computes the result of a sub problem and these results are then aggregated to provide the result of the entire computation.

Note the distinction between *parallelism* and *concurrency*. Parallelism refers to several threads cooperating to solve a computational problem while concurrency refers to several threads executing different computations that may be competing for shared resources. In this thesis we focus on applying parallelism to single task problems.

1.2 Terminology

This sections covers some of the terminology used in parallel programming. The terms defined here will be used throughout the thesis.

Race-condition: When a thread wishes to perform a computation on some data stored in main memory it has to be moved into a local register of the processor executing the thread. *Latency* refers to the time it takes from a processor requests a piece of memory until it is available. Both when reading into a register and when writing back to memory some latency is unavoidable. When considering only correctness this is not an issue in sequential programs. However, in parallel programs incorrect answers can occur if data access is done without any form of coordination. This is referred to as a race-condition. The classical example of a race-condition is a counter that is incremented by several concurrent threads. Assume we have two threads accessing some shared integer value, initialized to 1. If the two threads read the value at approximately the same time, both will read the value 1. Unaware of the fact that the other thread is also accessing this integer, each of the threads will load the value into a register and increment the value to 2. Since there is no coordination between the threads when accessing shared data the counter ends up with the wrong

value. It is clear that we need some way to prevent this from happening. One of the most prevalent forms of coordination is through mutual-exclusion

Mutual-exclusion: Write access to shared memory occurs in what is called a *critical section* of program code. Only one thread can occupy a critical section at a time. Every change made to shared memory should happen in a critical section. There are several different ways to implement critical sections and different properties to consider. The standard way of implementing mutual-exclusion is by using locks. When a thread enters a critical section it must first acquire a lock. If the lock is already held by someone else the thread waits until the lock is released. Critical sections should be used sparsely and be as short as possible. Sparsely because of overhead in acquiring and checking for locks; a lock should only be used to prevent race-conditions. It should also be short to minimize the time other threads have to wait to acquire a lock.

Dead-lock: When using multiple critical sections it is easy to imagine scenarios where a thread t_1 is holding a lock a waiting for lock b while thread t_2 is holding lock b waiting for lock a. Such situations are referred to as dead-locks. When a program enters a dead-lock the execution may halt indefinitely. There exists several useful heuristics for avoiding dead-locks. For instance, a thread should ideally never request a new lock while holding another, but if it holds several, locks should be released in the opposite order of which they where acquired.

Starvation: Every thread that attempts to acquire a lock should eventually succeed. If not, we say that the thread is starving. Note that no starving threads implies no dead-locks.

Fairness: If thread t_1 requests a lock before thread t_2 then t_1 should enter the critical section before t_2. We call this property the fairness of a locking scheme.

Scalability: By adding more processors one would expect the time to complete a computation would go down. When we say a problem scales well, we mean that our speedup is increasing as more processors are added. In mutual-exclusion approaches it is not uncommon for a problem to achieve speedup for a few processors only to stagnate or even diminish in performance as more processors are added.

Thread-safe: By thread-safe we mean a piece of code that can be accessed concurrently by several threads without risking race-conditions or invalid memory accesses. To make this clear we provide an example involving the STL data-structure `vector`. A vector is a list data structure that can store arbitrary types in sequential memory. The length of the list can grow and shrink dynamically to fit all its elements. This makes the vector one of the most versatile data structures found in the STL. It also makes it inherently not thread-safe. If a vector needs to grow to fit a new element all elements might have to be copied to a different location in memory. This happens because of the guarantee that all elements will be stored sequentially. Imagine what happens if one thread adds an element, or performs a `push_back` as it is called in the STL. This might trigger the whole vector to be copied if there is no room to grow in the current memory location. If another thread tries to access some element while

this is happening, we get what is in the C++ specification referred to as *undefined behavior*. This is used throughout the specification to signal that it is an illegal operation, but the exact behavior is left to the implementer to decide. Undefined behavior is always a bad thing. If we need to do any operation that alters the vector we need to do this in a mutual exclusive manner. Reasoning like this is important when doing parallel programming. If not, it might lead to program crashes and hard-to-find bugs. Even worse, the program might function perfectly fine for long periods of time without any error occurring, until one day when it does not.

Oversubscription: When writing programs to take advantage of available parallelism, it can be hard to create the correct amount of logical threads. Logical threads map to physical threads provided by the processor. At first glance this might seem trivial, all that is needed is to create as many logical threads as there are physical threads. This is easier said than done in cases where, for instance, threads are created in a loop or a recursive subroutine. Too many logical threads instantiated leads to overhead caused by context-switches. This is known as oversubscription. Another issue to watch out for is *undersubscription*. This occurs when there are fewer logical threads than there are physical threads. Of the two, oversubscription is by far the more likely problem.

1.3 Parallel patterns

Programming patterns were popularized with the release of the book *Design Patterns* [15]. By relying on commonly arising patterns programmers could use new abstractions that were easy to understand, both in code and when communicating ideas. As patterns for program design, there exists several reoccurring design patterns in parallel programming. A framework for doing parallel computing should encapsulate these patterns and provide the programmer with a way to express parallelism using higher level abstractions, without worrying about the underlying details. A good reference for parallel design patterns can be found in [21]. We will here look at some of the more common patterns. These patterns will be used extensively in the implementations presented later.

The most basic construct is the *parallel for*. This patterns splits iterations of a loop across available processing units. Write access to shared memory should be guarded by a lock. Sometimes we need to aggregate some result in a loop. For instance finding the sum over all elements in an array. A pattern for this operations is the *parallel reduce*. Again, the iterations of a for-loop are distributed over available threads. The difference here is that each thread holds a local variable, accumulating the values in its iteration space. Upon completion, the threads must synchronize and each local value must be combined to produce the final result. These patterns will be referred to as *looping constructs*.

A pipeline, or *producer/consumer* is a pattern where one or more threads produce

some data which is then processed by one or more threads (the consumers). This approach can be useful when working on streams of data where the data must be processed in discrete steps. An example is image or video processing.

1.4 C++

The algorithms presented in this thesis are implemented using the C++ programming language. In this section we will give a brief overview and history of the language. We will also cover template meta programming, a powerful language feature which enables libraries like TBB to extend the language capabilities. C++ is a system-level, statically typed, general-purpose programming language developed by Bjarne Strousrup starting in 1979 as an enhancement to the C programming language. Renamed to C++ in 1983, it is one of the most popular languages in use today. C++ was also the first language to popularize the object-oriented programming paradigm. It is sometimes referred to as a "middle-level" language since it offers access to both low level features and high level abstractions. The language is maintained by the *C++ standards committee* [9]. The current language standard was released in 2003. The next version of the language, dubbed C++0x, is expected to become the new standard in March 2011. A complete history can be found in [28]

The popularity of C++ grew partly from its backward-compatibility with C. This allowed new programs written in C++ to take advantage of legacy libraries. This strong connection with C also means that implementing OpenMP, a set of compiler directives provided to ease the development of parallel code, was fairly easy.

The C++ standard is divided into two parts, the core language and the *Standard library* which later got renamed the *Standard template library* when support for templates were added. Templates is a powerful abstraction for writing generic code where the actual data types gets decided at compile time. This removes code duplication. For instance, a container data structure has the same semantics regardless of which data types it is storing. Since the actual types used are inferred during compile-time, no run time penalties needs to be endured. By combining several advanced C++ features with templates, language extensions (like `remove_if`) can be implemented as libraries akin to lisp macros. Since its initial release, the language has seen usage in fields and to solve problems not originally envisioned. This has lead to a somewhat difficult to understand syntax, by humans and compilers alike.

1.5 Multicore processor architecture

When writing parallel programs, performance issues can arise because of the way memory is physically accessed. A modern processor can access several different layers of memory. In addition to the main computer memory, a processor typically has several areas of memory on the processor chip itself, known as a *multi level cache*.

A modern CPU, like the Intel i7 processor, has 3 levels of cache. These are referred to as the L1, L2 and L3 cache. There is a trade-off with caches where the purchase cost per byte of memory increases as the latency of memory look up decreases. From the L1 to the L3 caches we have strictly increasing size and therefore an increase in latency. Caches are useful in order to speed up access to commonly requested data. They do however, present some problems in the context of parallel computing.

When a processor core changes a value residing in a register, this change is not immediately seen by other cores. The L1 and L2 caches are typically private to a core and so, for the value change to become visible to other cores, the value needs to be written to shared memory. In addition, values are not retrieved one at a time. Instead, enough values are retrieved to fill a *cache-line*, which is 64 bytes on the Intel i7. When a write occurs to any element in a cache line, a synchronization of the whole line follows. This takes time. Even though cache synchronization is usually implemented in hardware, a parallel program should try to minimize the amount of data shared between cores to achieve good performance.

It is important to at least be aware of physical issues like cache latency when applying parallelism.Changes that need to be synchronized require a write to slower memory. If this is done often it can severely affect the performance of the parallel program.

2
Approaches to parallelism

In this chapter we present the two approaches used to express parallelism in this thesis. In the following sections the core principles behind each approach is covered, followed by how they are invoked in code. Simple examples are presented to show the syntax, and to give the reader a general feel for how parallel programs are expressed. The last section contains a brief comparison noting the advantages and disadvantages of each approach.

2.1 Intel Threading Building Blocks

Threading Building Blocks (TBB) is a C++ library developed by Intel to ease the development of multi-threaded applications. It is available as part of Intels commercial multi-core suite *Parallel Studio* and also as an open source library. This thesis focuses on the open source library. TBB is implemented strictly as a library, there is no extended language syntax or special compiler features needed. This means that the library should work on all parallel architectures and operating systems where a feature complete C++ compiler is available. The calling conventions of the library are similar to those in the C++ Standard Template Library (STL) and, like the STL, make heavy use of templates.

TBB revolves around the concept of a *task*. A task is a unit of independent work. The programmer expresses problems as tasks. At what time the tasks are executed is decided by the run-time system. In TBB this behavior is realized through the *task scheduler*. By focusing on specifying tasks instead of managing threads the programmer is relieved of issues such as load balancing. TBB also improves

portability by abstracting the details of the underlying system thread APIs. To guide the execution of tasks, the programmer can use one of many available *parallel algorithm templates*. These implement several recurring parallel patterns such as loop parallelization and pipelines. These patterns can also be combined to create complex parallel algorithms. Hence the name "building blocks". As explained in [25], these templates are optimized for tasks that are *non-blocking* and can be executed out of order. For blocking or dependent tasks the programmer can control the task scheduler directly. An example of which will be shown in Chapter 3.

Threading Building Blocks is inspired by previous work in the field of parallel computing. Cilk, a general purpose programming language designed for parallel computing, introduced the concept of *work stealing* and *recursive range splitting* [8]. It showed that while sometimes slower on sequential machines, recursive ranges often leads to performance advantages regarding load-balancing and cache reuse. TBB also takes many design cues from the C++ standard template library [22]. Version 1.0 was introduced in 2006 [31], one year after the release of the first main stream dual-core x86 processor, the Pentium D [16]. There has been previous work showing the merits of TBB, including [3].

For scheduling, TBB uses a recursive divide and conquer approach. Tasks can be broken down into smaller pieces as required to take advantage of available parallelism. This decision was influenced by the Chare Kernel [27] which showed how splitting a program into many small tasks makes it easier to distribute across threads than if only a few large chunks are used. These tasks are then mapped to logical threads and executed as needed by the task scheduler. If cores become idle, the task scheduler may split a task and distribute a part to an idle thread. This mechanism is known in TBB as *task stealing*.

Upon instantiation, the task scheduler initializes several logical threads. Note the distinction between a logical thread, specified with a threading API, and a physical thread executing on a processor. These logical threads are placed in a *global thread pool*. By utilizing a thread pool, the scheduler removes the overhead of thread allocation and destruction between each parallel region.

TBB also offer a set of thread-safe containers. Most C++ developers rely on the containers implemented in the STL for storing data. While STL containers are optimized for performance and are in general very efficient, some are also inherently not thread-safe. The containers offered by TBB have the same semantics as their STL counterparts, but with added mechanisms to make them thread-safe. Using such containers offers a trade-off in performance for correctness. The concurrent containers implemented in TBB are *vector*, *queue* and *hash map*.

2.1.1 Example: parallel average

In this section we present a simple parallel program using TBB. The code implements an embarrassingly parallel problem, finding the average of three adjacent elements

```
1 struct Average {
2   float* input, output;
3   void operator()( const blocked_range<int>& range) const {
4     for( int i = range.begin(); i != range.end(); ++i )
5       output[i] = (input[i-1] + input[i] + input[i+1])/3.0f;
6   }
7 };
```

Listing 2.1: Parallel Average Task

```
1 void ParallelAverage( float* input, float* output, int n ) {
2   Average avg;
3   avg.input = input;
4   avg.output = output;
5   tbb::parallel_for( tbb::blocked_range<int>( 0, n, 1000 ), avg );
6 }
```

Listing 2.2: TBB template invocation

for all elements in a sequence. All elements are stored in sequential memory.

The example takes two arrays of floating point values, I and O, denoting the input and output respectively, and stores in element O_i the value $(I_{i-1}+I_i+I_{i+1})/3$. The example is taken from [25]. Note that for this example to execute correctly, the arrays have to be padded with one element at the front and at the end.

Listing 2.1 shows the task body we want to execute. A task body is realized in code as a C++ *function object*. A function object, or a *functor*, is a design-pattern used in programming languages that do not support higher-order functions. A detailed explanation can be found in [24]. Functors are useful for passing functions as parameters to other functions, known as higher order functions, for execution at a later time. This is often referred to as a callback or delegate function. Local data stored in a task instance is defined as regular member fields of the object. In this case, each instance of the task body holds two pointers to a `float` value. Serving as an entry point for the parallel template is the overloaded "()" operator. Overloading the () operator gives the functor the same calling convention as a regular C function. When executed, the task body receives a parameter referred to in TBB as a range. This can be thought of as a pair of iterators, giving the starting and ending indexes of the array elements assigned to be processed by this task instance. In this case the type of the range is `blocked_range<int>`. This is a built in range type for one dimensional iteration spaces. Lines 4 and 5 iterates over the range assigned to this task instance, retrieves values from the input array, performs the calculation and writes the results to the output array.

Listing 2.2 shows how the three pieces, task body, range and algorithm template fit together. First an instance of the task body is created and its pointer fields

assigned. On line 5, the parallel algorithm template is called, passing in a range and a task body instance. Behind the scene, the template spawns new task instances with the `Average` task body as payload. Notice the strong but subtle distinction between a task and a task body. A task implements splitting and execution logic, while the task body implements the computation itself. When using algorithm templates the former is handled by the template.

The third parameter passed to `blocked_range` defines a *grainsize*. The grainsize sets a threshold for the minimum size of a sub-range. Note that by default this value is only guiding, the task scheduler will not create more tasks than needed to utilize available parallelism. When splitting occurs, the range is divided into two equal parts with any rounding errors going to the first. Determining the ideal size of the grainsize is more an art than a science and requires experimentation, but a recommended heuristic is to split a range so that a task takes at least 10,000 machine instructions to complete. Improving performance through optimizing grainsize is an ongoing research topic covered in [26].

We have now seen an example of a typical usage pattern for Threading Building Blocks. The steps described in the example above, where the programmer provides task bodies and ranges to templates, will be a recurring theme in this thesis. It is therefore important to be comfortable with the definitions given here.

2.2 OpenMP

As mentioned, parallel computing is not a new phenomenon. Already in the 1970's several vector- and parallel computers were used. Support for programming these machines came in the form of language extensions to popular languages, first Fortran and later C. Each computer vendor provided their own extensions with similar semantics but different syntax. Because of this, moving programs between platforms was non-trivial. This problem only increased onward into the eighties and nineties. Several attempts where made to standardize these extensions with a varying degree of success. In April of 1996, SGI; a manufacturer of high-performance computers, bought one of its rivals Cray. Since the software tools for the two companies platforms were not compatible a new extension had to be specified. Not wanting this problem to arise every few years, SGI spearheaded the forming of the Architecture Review Board (ARB). The goal of this board was to write an open specification for parallel computing on shared-memory systems. Eighteen months later, in October 1997, OpenMP was born. The standard is backed by several large computer companies like Intel, IBM, Compaq and Silicon Graphics, and is today the *defacto* standard for parallel programming on shared-memory systems. The OpenMP specification [23] is currently in its third major revision.

The OpenMP Application Programming Interface (API) enables portable shared-memory parallel programming in C/C++ and Fortran. It consists of a set of com-

piler directives, library routines and environment variables. OpenMP has seen widespread adoption and is implemented by most popular compilers like the GNU Compiler Collection (GCC), Intels C/C++ compiler (ICC) and the Microsoft C/C++ compiler.

OpenMP allows a section of code to be executed by several cooperating threads. This is done using the "Fork and Join" programming model [12]. A *master* thread forks and creates several *worker* threads. These threads then performs a series of instructions in parallel which make up a *parallel region*. At the end of the parallel region the worker threads waits until all threads have finished and are then killed off. After this, sequential execution is continued by the master thread. The astute reader may notice that the only high-level difference from a thread pool approach is the termination of worker threads after the parallel region

The programmer specifies which sections should be executed concurrently by inserting *directives*. In C/C++ any line that starts with a # sign is interpreted as a command for the preprocessor. A preprocessor is defined as a program than alters its input in some way before passing it as output to some other program, in this case the compiler. An example of this is the `#include` directive. When the preprocessor encounters a line like `#include <stdlib.h>` it replaces the line with the content of the file named in brackets, in this case `stdlib.h`. All OpenMP C/C++ directives start with `#pragma omp`, followed by one or more clauses. The initialization and destruction of the worker threads is handled by the compiler. This high-level specification can turn a sequential program into one that utilizes multiple processors fairly easily. However, to get a satisfactory performance increase, some optimization and code restructuring is often necessary. Nonetheless, this approach of turning a sequential program into one that can execute in parallel in incremental steps, is one of the most appealing features of OpenMP.

OpenMP also includes several helper functions. These functions let the programmer manage threads at run-time. The most useful of which allows querying a thread for its *id* and to get the total number of available threads.

OpenMP was created as a way to unify the myriads of directive based dialects that were common in the field of high-performance computing. When it was created this mostly meant ways of parallelizing loops used for numerical calculations. More recently, the sophistication required by programmers from their threading packages have increased. Several types of parallelism used to be hard to express with OpenMP, like recursive parallelism. While not impossible, such types of parallelism used to require a substantial effort and hand-tuning by the programmer to yield efficient implementations. This would defeat some of the main selling points of OpenMP, the speed of which parallel code can be written. With version 3.0 of the OpenMP specification, support for a tasking model has been added. Like in TBB, this allows the programmer to specify tasks and delegate time of execution to the run-time system. Chapter 3 presents a mergesort-algorithm implemented using this task construct. An evaluation of the OpenMP tasking model can be found in [5].

```
1  void ParallelAverage(float* input, float* output, int n) {
2  #pragma omp parallel for
3    for(int i = 1; i < n; ++i) {
4      output[i] = (input[i-1] + input[i] + input[i+1]/3.0f;
5    }
6  }
```

Listing 2.3: OpenMP Parallel Average

2.2.1 Example: parallel average

The example presented here implements the same semantics as in Section 2.1.1. An array of floating point values, a sufficiently large output buffer and the length of the input is passed in as parameters. We then proceed to calculate the average over $input[i-1] + input[i] + input[i+1]$.

By looking at Listing 2.3 it is clear that OpenMP has a decidedly lower barrier of entry than TBB. The only change of the sequential code needed to make it run in parallel is to add the pragma directive on line 2. `#pragma omp` marks the start of a parallel region, and inside a parallel region we can request that code be executed in parallel. This is called a *parallel construct* in OpenMP terms. The parallel construct used here is a parallel for, which distributes iterations over available threads. Note that the syntax for a parallel for construct is `#pragma omp for`. The pattern of declaring a parallel construct immediately after a parallel region is so common that the two can be combined into a single statement. OpenMP supports several different constructs and predicates, all of which are relatively easy to understand. These will be covered as needed when explaining code listings. For complete coverage of the functionality provided by OpenMP [12] is recommended.

This example shows how little effort is required when retrofitting loops to run in parallel with OpenMP. Of course, this is a trivial example. Given more involved algorithms, code analysis has to be performed to identify potential race-conditions. Code may also need to be restructured to optimize it for multiple threads.

2.3 Comparison

OpenMP and TBB are two potential solutions to the same problem, enabling the programmer to construct correct programs that utilize parallelism without resorting to manual thread management. The differentiating factors are performance, level of abstraction and ease of use. In this section we will try to give the reader an understanding of the major differences between the two approaches.

Availability: OpenMP needs to be implemented in the compiler. Most popular compilers implement support for the OpenMP specification, the only notable exception is Clang, a C/C++ front-end for the *Low-level virtual machine*(LLVM)

```
1  for (int i = 0; i < n; ++i) {
2    if(i<n/2){ DoCubedWork(); }
3    else{      DoLinearWork();}
4  }
```

Listing 2.4: A loop with unevenly distributed workload

compiler. Being so closely tied to the compiler yields some advantages. Foremost one can be reasonably sure a program will compile on most machines. Another benefit inherited in the specification is that OpenMP pragma statements not supported by the compiler will be ignored. This means that the programs will compile even with compilers that do not support OpenMP. TBB, on the other hand, is a C++ library. This means that the library has to be installed for a program to take advantage of it. In addition, the TBB shared-library (or DLL on Windows) has to be available on the machine that runs the program. While this can cumbersome, it also means that no special compiler support is needed, since TBB is written in C++.

Load balancing: Another benefit of TBB is the way load-balancing is achieved. We will make this point clear by means of an example. The loop in listing 2.4 shows that the amount of time taken by each iteration is not equal. `DoCubedWork()` runs in time $\theta(n^3)$ and `DoLinearWork()` runs in time linear in n. Clearly, most of the time spent in this loop will be in iterations 0 to $n/2$. With OpenMP, a programmer would first attempt parallelizing this loop by inserting a `parallel for` pragma.

```
#pragma omp parallel for
```

This would split the work among available threads. In TBB one would have to include the loop in a function object, the task body. As an experiment, the functions shown in Listing 2.4 where implemented in both OpenMP and TBB. The results printed below are from execution on a machine with an Intel Core 2 Duo Processor which supports two concurrent threads.

```
Using tbb :   Time elapsed: 0.877825 seconds.
Using omp :   Time elapsed: 1.65978 seconds.
Sequential:   Time elapsed: 1.6593 seconds.
```

When using OpenMP, the iterations get evenly distributed among the available threads. This means that thread 0 gets iterations 0 to $n/2 - 1$ and thread 1 gets iterations $n/2$ to n. Notice that there is no speedup by invoking OpenMP, in fact it is slightly slower. This is because when a thread finishes is has to wait for all other threads in the parallel region to finish. In the example, thread 1 is finished much faster than thread 0. If we look at the results for TBB we see that we achieved significant speedup. This is due to task stealing. In the example given, we could

easily fix the OpenMP performance by using *dynamic scheduling*. Doing this is very simple, the programmer just has to append `schedule(dynamic)` to the pragma. With dynamic scheduling threads will request additional iterations after running *chunksize* number of iterations, where chunksize is specified by the programmer.

Remember that in TBB, the programmer specifies tasks, not threads. The exact number of tasks created is handled by the task scheduler. After a core has finished all of its tasks, then if there are tasks queued up at another core that is still processing a task, the thread can "steal" a queued task. So while the TBB code takes longer to implement up front, we gain performance because the implementation has more flexibility at run-time.

Implementing: While TBB requires more code than OpenMP and programs often have to be written from scratch to adhere with the design of the library, there are some clear cut advantages to using TBB over OpenMP. First of, it offers a higher level of abstraction. Where a *parallel region* in OpenMP can only operate on primitive values and arrays, TBB works with any thread-safe data structure. Secondly, TBB allows more complex parallelism to be expressed. The programmer is free to use the task scheduler directly to create their own parallel patterns, while OpenMP is limited to the built in constructs. The open source TBB library also offers slightly better support during development. For instance, assert statements are provided to catch common programming errors.

In OpenMP a programmer has the choice of one of three scheduling techniques, static, guided and dynamic. TBB replaces this with a single, automatic, divide and conquer technique. This technique has been shown to be superior to guided and dynamic scheduling, but beaten by a finely tuned static scheduler [25].

The greatest advantage of OpenMP is the small amount of changes needed to utilize parallelism in existing code. This can make it a better choice when retrofitting an existing code base. In professional development, time is often a scarce resource, the amount of time needed to add `pragma` statements is drastically less than refactoring computations into task bodies.

Another, maybe minor, point in TBBs favor is that it is written in a programming language. Since OpenMP is a set of directives it is not immediately obvious to a programmer with no experience using the approach. Simple constructs may easily convey their semantics but a complex directive containing several clauses and a scheduling strategy may be harder to grok. TBB on the other hand is written in pure C++. This means that a programmer who thoroughly understands the language semantics, which is in no way a small feat, will understand the intention.

The two approaches are not mutually exclusive. TBB, being the newer of the two, was designed to not interfere with any OpenMP threads. In fact, OpenMP can be used inside TBB code and vice versa. This does however suffer a performance penalty.

3 Mergesort

In this chapter we take a more in-depth look at the parallelization approaches by implementing a sorting algorithm, Mergesort. This algorithm was chosen because it is a basic algorithm that most readers should already be familiar with. The recursive nature of the algorithm also makes it an ideal candidate to highlight the differences between an effective sequential and parallel implementation. The following section presents an overview of the sequential algorithm, as well as some background information and research activity. After this comes a section explaining the optimization techniques used in the implementation. The section then goes on to explain how the algorithm can be made parallel. After this comes a section detailing the code itself. The chapter ends with a section presenting experimental results and a discussion.

3.1 Overview

Mergesort is a comparison based, stable sorting algorithm with run-time complexity $O(n \log n)$. It is covered in most introductory books on algorithms. A good example of which is Cormen *et al.* [10]. Sorting, being one of the fundamental operations used in programs, has been extensively covered in the literature. In [7], Bitton presents a taxonomy of parallel sorting algorithms which include several parallel Mergesort algorithms on shared-memory systems. Parallel mergesort algorithms are in wide-spread use today being, particularly useful in online sorting.

The algorithm sorts a sequence of comparable objects by splitting the input into two pieces and then solves each by applying recursion. The invocations of the recursive calls form a balanced binary-tree with depth $\log n$, where n is the length

of the sequence. After the tree has been expanded, a parent node merges the sorted sequences from its two children with the Merge algorithm. The Merge algorithm defines pointers to the start of each sorted sequences given as input. It then proceeds to store the element of whichever pointer points to the smallest element in a new list, after which the pointer is incremented. This is continued until all pointers have traversed their respective input sequences, yielding a combined sequence of all input sequences in sorted order. The recursive Mergesort algorithm is shown as Algorithm 1. It is invoked by passing the sequence to be sorted as well as pointers to the start and end of the sequence, denoted in Algorithm 1 as A, p and r respectively.

Algorithm 1 Recursive mergesort

```
MergeSort(A, p, r)
if p < r then
    q = floor((p + r)/2)
    MergeSort(A, p, q)
    MergeSort(A, q + 1, r)
    Merge(A, p, q, r)
end if
```

Mergesort belongs to a set of algorithms known as *Divide and conquer* algorithms. This makes it an ideal candidate for increased parallel performance since the partitioning of the problem into independent sub problems comes naturally. The recursive nature of algorithms makes it problematic to express parallelism using a parallel loop construct. This is highlighted in [29].

3.2 Analysis

Here we present two well known optimizations that can be applied to Mergesort. First, iterative languages are able to process data iteratively more efficiently than recursively. This has been studied extensively in the literature [19]. By turning the recursive implementation into an equivalent iterative implementation, we can eliminate the overhead of stack frames during execution. This will be referred to as *recursive flattening*. A recursive Mergesort algorithm can be run iteratively by sorting sub-sequences of length 2^i, starting from 2^0 and ending with $2^{\log n-1} = \frac{n}{2}$. This assumes that n is a power of two.

Mergesort requires a temporary buffer to hold the sorted sequence produced by the Merge algorithm. In a naive implementation, this buffer is created for every invocation of Merge. This can be optimized by allocating the buffer once, at the start of the algorithm. For each of the $\log n$ passes over the sequence, the input- and temporary buffer are then switched. This optimization will be referred to as *double buffering*.

3.2.1 Exploiting parallelism

Mergesort, being a divide and conquer algorithm, parallelizes quite easily. Observe that $MergeSort(A, p, q)$ and $MergeSort(A, q+1, r)$ in Algorithm 1 can be executed in parallel. This is a well known fact and is presented in, among others, [32]. Each node in the recursion tree has a dependency on its children. This limits the amount of parallelism that can be utilized near the root of the tree. At each depth d, $0 < d < \log(n)$, there are 2^d non-overlapping sequences. This means that at any given depth at most $m = 2^{d-1}$ threads can be utilized. To improve this, a *Parallel Merge* can be applied to levels where $m < p$, with p referring to the number of processors available. The technique is presented by Reinders in [25]. Figure 3.1 illustrates the algorithm. Given two sorted sequences, s_1 and s_2, to be merged, we define m_1 as a pointer to the middle element of s_1, and m_2 as a pointer to the position m_1 would be inserted into s_2 by a insertion based sorting algorithm. This splits s_1 into two independent pieces a and b, where all elements in a are less than or equal to m_1 and all elements of b are greater than m_1. c and d are defined similarly for s_2. a and c can now be merged on one processor, while b and d gets merged on another in parallel. The algorithm can also split the sequences further if more than two processors are available. After all merges are complete the sequences are concatenated to yield the complete sorted sequence. The efficiency of the algorithm depends on m_2 splitting s_2 into similarly sized parts.

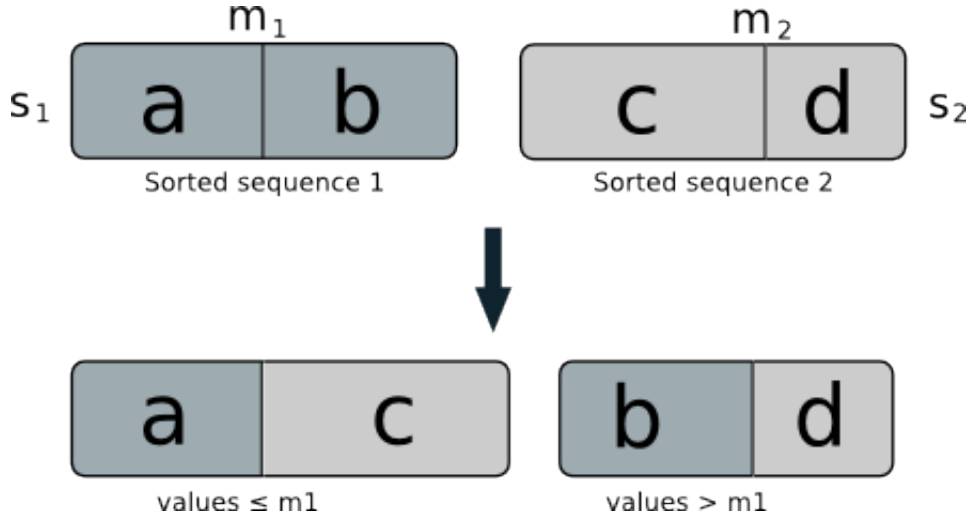

Figure 3.1: Illustration of splitting two sorted sequences into four that can then be merged in parallel.

For sufficiently small sequences, sequential sorting is more efficient than a parallel algorithm. As the input is split into smaller and smaller parts by recursive invocations of Mergesort, a cutoff can be introduced. For input smaller than this

cutoff, sequential sorting will be applied. An algorithm using this technique will be referred to as *small sequence sequential*.

3.3 Implementations

This section covers the implementation of the Mergesort algorithms. In order to compare the speedup of each parallel algorithm, a sequential, iterative Mergesort algorithm was implemented. This was optimized with recursive flattening and double buffering. To demonstrate how easily a loop based algorithm can be made parallel with OpenMP, it makes use of an optional parallel for directive. We cover this algorithm in Subsection 3.3.1.

In Subsection 3.3.2 and Subsection 3.3.3 we present parallel recursive mergesort algorithms implemented in OpenMP and TBB respectively. Both of these algorithms make use of a Parallel Merge to utilize available threads, the implementation of which is covered in Subsection 3.3.4.

3.3.1 Iterative mergesort

```
1  int* MergeSort(int* a, int n, bool use_omp) {
2    int log_n = log2(n);
3    int *scratch = (int*) malloc(n * sizeof(int));
4    for (int i = 0; i < log_n; ++i)
5      {
6        int interval = 2 << i;
7
8        #pragma omp parallel for if (use_omp)
9        for (int j = 0; j < n; j += interval)
10         {
11           merge(a+j, a+j+interval, scratch+j);
12         }
13       std::swap(a, scratch);
14     }
15   free(scratch);
16   return a;
17 }
```

Listing 3.1: Iterative mergesort with optional OpenMP directive

Listing 3.1 shows an iterative mergesort algorithm. The algorithm uses $\log n$ passes over the input to produce a sorted sequence. Each of these passes can be done in parallel. This is achieved by adding an OpenMP parallel for directive. This also demonstrates the first use of an OpenMP `clause` in this thesis, an if clause. It evaluates the expression given in parenthesis. If the expression evaluates to true, the loop following the directive is made to run in parallel. This is included to achieve two things. First, the algorithm will be timed and used without the parallel region as a

base time from which the speedup of the parallel algorithms is calculated. Second, we will see how much speedup can be achieved with minimal effort. Whenever a nested loop is parallelized it is always best to apply the parallel directive to the outermost loop. By doing this, threads do not have to be created and destroyed more than once. In this case, iterations of the outer loop have to be performed in sequence to get the correct result, so this is not possible.

```
1  void inline merge(int* start, int * const end, int* scratch){
2    // Get pointers to the start of each sequence
3    int* fst_seq = start;
4    int* sec_seq = start + (int) (end - start)/2;
5    const int* middle = sec_seq;
6
7    // Calculate the length of the combined sequences
8    int len = (int) (end - start);
9
10   // Take min(fst_seq, sec_seq), add to scratch and increment
11   for (int i = 0; i < len; ++i) {
12     // If elem in fst_seq is smaller and fs_seq has more elems
13     // or sec_seq is empty
14     if( ( *(fst_seq) <= *(sec_seq) && fst_seq != middle )
15       || sec_seq == end)
16     {
17       *(scratch++) = *(fst_seq++);
18     } else {
19       *(scratch++) = *(sec_seq++);
20     }
21   }
22 }
```

Listing 3.2: The merge function used in Mergesort (Listing 3.1)

As input the algorithm takes a pointer to an array of integers and its length. On line 3 the temporary buffer, `scratch`, is allocated. Each iteration of the outer for loop represents the merge performed on a given depth of the recursion tree. The variable `interval` define sub-sequences of increasing length, from 2^0 to $2^{\log n-1}$. Each of which are merged by the helper routine `merge` in the inner for loop. The code for `merge` is shown in Listing 3.2. `merge` merges the sequence defined by the two first parameters into the buffer given by the third using a standard merge implementation. Since the two sequences are assumed to be stored one after the other in adjacent memory and of equal size, we only need two input pointers. The length of each sequence is then calculated as the distance between the pointers divided by two. Finally, the input and temporary buffers are swapped for each iteration of the outer for loop.

3.3.2 Recursive OpenMP Mergesort

We now turn to a recursive parallel Mergesort algorithm, implemented using the tasking model introduced in OpenMP 3.0. Listing 3.3 shows the top-level algorithm that serves as the entry point to the recursive `merge_sort` function. After a temporary buffer has been allocated a parallel region is defined. This directive tells the compiler to emit code to create worker threads. Inside the parallel region another OpenMP directive containing two clauses, single and `nowait`, is defined. The `single` clause signals that the following code is to be executed by one thread only. By default, any thread not executing the function will idle until the call completes. This is not what we want as worker threads will be assigned work inside `merge_sort`. By adding the `nowait` clause this behavior is overridden. The `merge_sort` function takes five parameters, pointers to the start and end of the two buffers and a pointer to an integer. The last parameter is used to determine which of the two buffers contain the final sorted sequence.

```
1  int* OMP_MergeSort_Task(int* data, int n) {
2    int *scratch = (int*) malloc(n * sizeof(int));
3
4    int dirval = -1;
5    #pragma omp parallel
6    {
7      #pragma omp single nowait
8      merge_sort (data, data+n, scratch, scratch+n, &dirval );
9    }
10   if(dirval == 1){
11     free (scratch);
12     return data;
13   }
14   free (data);
15   return scratch;
16 }
```

Listing 3.3: Top-level OpenMP Mergesort function

Listing 3.4 shows the recursive Mergesort algorithm. The algorithm starts by checking if sorting should be done sequentially. Sequential sorting is done for sequences with less elements than some `CutOff`, the exact value of which will be discussed in Section 3.4. If the input contains more elements than the cutoff value, the buffers are swapped and pointers to the middle of the input sequences defined. After this, `merge_sort` is called recursively for each sub-sequence, the first of which is decorated with a task directive. The OpenMP task directive declares the function immediately following to represent a new task. Notice that in each recursive invocation of `merge_sort` only one new task is created. The second recursive call, shown on line 16, gets executed by the current thread. After this comes a directive with a `taskwait` clause. This clause acts as a synchronization barrier, ensuring that the

```
1  void merge_sort(int* begin, int* end, int* begin2, int* end2, int* direction)
       {
2    int n = end - begin;
3    if(n <= CutOff) {
4      std::sort(begin, end);
5    }
6    else {
7      if(*direction == -1) {
8        std::swap(begin, begin2);
9        std::swap(end, end2);
10     }
11     int* middle = begin + (end-begin)/2;
12     int* middle2 = begin2 + (middle - begin);
13 #pragma omp task
14     merge_sort(begin, middle, begin2, middle2, direction);
15
16     merge_sort(middle, end, middle2, end2, direction);
17 #pragma omp taskwait
18     if ( n >= pmCutoff) {
19       ParallelMerge(begin, middle, middle, end, begin2);
20     } else {
21       std::merge(begin, middle, middle, end, begin2);
22     }
23   }
24   *direction *= -1;
25 }
```

Listing 3.4: OpenMP mergesort with tasking

two recursive calls have run to completion before continuing.

After the barrier on line 17, a check is performed to see if the sequence is large enough to justify a Parallel Merge. The cutoff value used here, `pmCutOff`, is defined as the size of the whole sequence to be sorted divided by the number of threads available. This allows all threads available to be utilized in the merge. The `ParallelMerge` function is implemented using TBB. This shows that OpenMP and TBB can be used together. We leave the discussion regarding if this is efficient or not to Section 3.4, where we present the results.

If the input is less than the cutoff, a sequential merge is done instead. Note that `std::merge` is not the merge routine shown in Listing 3.2, but rather the Merge algorithm provided by the STL.

3.3.3 TBB Mergesort

Listing 3.5 shows the top level of the TBB Mergesort implementation. Because of the dependencies between tasks, a parallel algorithm template cannot be applied. Instead the algorithm implements a new task, `MergeSortTask`. On line 5, the task object gets allocated using a special `new` operator provided by TBB. The operator is parameterised on the type of task being allocated, in this case a root task. This enables task hierarchies to be declared. The task constructor takes two pairs of iterators as input. These iterators point to the beginning and end of their respective sequences. On line 8 the task scheduler is instructed to spawn the newly allocated task with a call that blocks until the task has completed.

```
1 template<typename Iterator>
2 void TBB_MergeSort( Iterator begin, Iterator end, Iterator out, Iterator
      out_end)
3 {
4   // Allocate a new root task
5   MergeSortTask<Iterator>& a = *new(task::allocate_root())
6     MergeSortTask<Iterator>(begin, end, out, out_end);
7   // Spawn the task and wait for it to terminate.
8   task::spawn_root_and_wait(a);
9 }
```

Listing 3.5: TBB mergesort task

When a task is spawned, the task scheduler calls its `execute` method. This happens from within the TBB library. Listing 3.6 shows the implementation of the `execute` method for `MergeSortTask`, an object which inherit from an abstract baseclass `tbb::task`. Once called by the task scheduler, the method first checks if the input should be sorted sequentially. For inputs larger than the cutoff, the two buffers are swapped and two child tasks get allocated and initialized. The thread scheduler keeps track of allocated tasks via reference counting. Reference counting is a common idiom in languages that do not offer garbage collection. It is used to keep track of how many objects hold a reference to the object in question. Once a reference counter reaches 0, the referee can no longer be accessed and can therefore be destructed. This is set on line 22 through the method `set_ref_count` inherited from the `task` base class. The count is set to 3, since the parent has to be included. The children are then spawned and the parent waits for them to complete. The tasks are then spawned and the method waits for the child tasks to complete. After all child tasks are finished, a parallel merged is formed for long sequences. As with the recursive OpenMP algorithm covered in Subsection 3.3.2, the cutoff value, `pmCutOff`, is defined as the size of the complete sequence to be sorted divided by the number of threads available.

Upon completion, the method returns `NULL` to signal to the task scheduler that the task is completed, which decrements the reference count by 1. The count will

```
1  class MergeSortTask : public tbb::task
2    // ... Member declaration and ctors here.
3    task* execute()
4    {
5      int n = end - begin;
6      if((end-begin) <= CutOff)
7        std::sort(begin, end);
8      else {
9        if(*direction == -1){
10         std::swap(begin, begin2);
11         std::swap(end, end2);
12       }
13       // Get iterator for middle of both arrays
14       Iterator middle  = begin + (end-begin)/2;
15       Iterator middle2 = begin2 + (middle-begin);
16       // Allocate space for child nodes
17       MergeSortTask& a =  *new( allocate_child() )
18         MergeSortTask(begin, middle, begin2, middle2);
19       MergeSortTask& b =  *new( allocate_child() )
20         MergeSortTask(middle, end, middle2, end2);
21       // Set reference count
22       set_ref_count(3);
23       // Spawn tasks and wait.
24       spawn( b );
25       spawn_and_wait_for_all( a );
26
27       if ( n >= pmCutoff) {
28         ParallelMerge(begin, middle, middle, end, begin2);
29       } else {
30         std::merge(begin, middle, middle, end, begin2);
31       }
32     }
33     return NULL;
34   }
35 };
```

Listing 3.6: The execute method of `MergeSortTask`. Constructor and member declaration have been omitted to save space.

reach 0 as the root task (allocated in Listing 3.5) terminates.

3.3.4 Parallel Merge implemented with TBB

The top level code for `ParallelMerge`, shown in Listing 3.7, is similar to the example shown in Listing 2.2, in that both use a `parallel_for` algorithm template. As mentioned, a `parallel_for` template takes two parameters, a range; specifying how the input data is to be split among task instances and a task body; the actual computations to perform. While the example in Subsection 2.1.1 made used of a `blocked_range`, which is provided as part of TBB, this algorithm uses a custom range, `ParallelMergeRange`. The implementation of which is shown in Listing 3.8. The `struct ParallelMergeRange` is taken from a reference book on TBB [25]. In the book, Parallel Merge serves as an example of input distribution using a non-trivial range object. When deciding if a new task instance should be created, the task scheduler checks if the input can be divided any further is divisible. A custom range therefore needs to provide the methods `empty()` and `is_divisible()`, shown on lines 5 and 6. If `is_divisible` returns true, the task scheduler invokes a special constructor, known as a *splitting constructor*. The splitting constructor implements the logic for instantiating a new range, which takes half the data covered by the range passed in as parameter. The second parameter is an object of type `tbb::split`, the sole purpose of which is to distinguishing a splitting constructor from the C++ copy constructor. The constructor performs pointer arithmetic to split the sequence as explained in Section 3.2. Notice that elements never get copied. This makes range splitting very efficient.

```
1 template<typename Iterator>
2 void ParallelMerge( Iterator begin1, Iterator end1, Iterator begin2, Iterator
      end2, Iterator out ) {
3   parallel_for(ParallelMergeRange<Iterator>(begin1,end1,begin2,end2,out),
4                ParallelMergeBody<Iterator>());
```

Listing 3.7: Top-level Parallel Merge function.

Listing 3.9 shows the implementation of the task body, which is quite trivial, since the logic interesting parts has to do with how the data gets split. Given a range, the body passes iterators stored in the range object to the `std::merge` function.

```
template<typename Iterator>
struct ParallelMergeRange {
  static int grainsize;
  Iterator begin1, end1; // First input sequence
  Iterator begin2, end2; // Second input sequence
  Iterator out;          // Output buffer
  bool empty() const {return (end1-begin1)+(end2-begin2)==0;}
  bool is_divisible() const {
    return std::min( end1-begin1, end2-begin2 ) > grainsize;
  }
  // Splitting constructor
  ParallelMergeRange( ParallelMergeRange& r, tbb::split ) {
    // Iterator to middle of r's first sequence.
    Iterator m1 = r.begin1 + (r.end1-r.begin1)/2;
    // Get an iterator to where *m1 would be inserted
    // into r's second sequence by insertion sort.
    Iterator m2 = std::lower_bound( r.begin2, r.end2, *m1 );

    // Assign member iterators
    begin1 = m1;
    begin2 = m2;
    end1 = r.end1;
    end2 = r.end2;
    // Our output buffer comes after r's
    out = r.out + (m1-r.begin1) + (m2-r.begin2);

    // Update r's pointers to reflect changes
    r.end1 = m1;
    r.end2 = m2;
  }
  // Constructor
  ParallelMergeRange( Iterator begin1_, Iterator end1_,
                      Iterator begin2_, Iterator end2_,
                      Iterator out_ ) :
    begin1(begin1_), end1(end1_),
    begin2(begin2_), end2(end2_), out(out_) {}
};
```

Listing 3.8: The range used in Parallel Merge

```
template<typename Iterator>
struct ParallelMergeBody {
  void operator()( ParallelMergeRange<Iterator>& r ) const {
    std::merge( r.begin1, r.end1, r.begin2, r.end2, r.out );
  }
};
```

Listing 3.9: The task body used in Parallel Merge.

3.4 Tests setup and results

The experiments were conducted on a machine with 2 six-core AMD Operton 2431 processors (2.4 GHz) with 8 GB of memory. The machine was running Linux with kernel version 2.6.18. The program was compiled with GCC version 4.5.2 using the O3 optimization level. Version `tbb30_20100406oss` of the TBB library was used.

For each test five different arrays of random integers where generated. This was done to reduce the likelihood that a particular array instance would have some order that would make some algorithm come out favorably. Set of five arrays with length ranging from 2^{19} to 2^{28} were then created, initialized with random integers. The random numbers were generated with the srand function from the C standard library. The running times given is the time an algorithm used to sort the arrays divided by the number of arrays. Each algorithm got the same five arrays as input.

In addition to varying the length of the input, all algorithms were tested with 1,2,4,6,8,10 and 12 cores enabled. The cutoff below which the two recursive algorithms would resort to sequential sorting was set to 2^{16}. Parallel merges was used for sequences larger than the size of the input divided by the number of cores used.

The TBB library includes a sorting routine, `parallel_sort`. This was included in the evaluation to see how the algorithms presented here measure up to a parallel sorting routine in wide-spread use. From [25], `parallel_sort` is a non-recursive quicksort algorithm, which has a run-time complexity not exceeding $O(n\,log(n))$ on a single processor and approaches $O(N)$ as more processors are used.

3.4.1 Results

We will now look at the result of the experimental evaluation. Table 3.1 shows the average execution time of all algorithms presented. The two first entries, *It.MS* and *OMP It. MS*, refer to the iterative algorithm presented in Subsection 3.3.1. Following those two are *OMP MS* and *TBB MS*; the recursive implementations covered in Subsection 3.3.2 and 3.3.3 respectively. The last entry, *TBB Sort*, refers to the parallel quicksort algorithm provided by TBB. The fastest run-times for each input size is marked with green.

The recursive Mergesort algorithms compare favorably to TBB Sort, beating the algorithm in all runs for input of length 2^{25} and above. This can also be seen in Figures 3.4 and 3.5. The speed difference between the algorithms varies widely based on the size of the input and how many cores where enabled. Figure 3.2 visualizes this. Note especially the performance of TBB Sort and OMP MS here. The performance of the two algorithms vary a great deal when using different numbers of threads, with TBB Sort offering little speedup when going from 4 to 6 enabled cores, while OMP MS performance increases greatly. This then changes when going from 6 to 8 cores, where TBB Sort sees a performance boost not seen in OMP MS. This is most likely caused by how efficient the work gets distributed because of the grainsize.

Algorithm	2^{24}	2^{25}	2^{26}	2^{27}	2^{28}
`It.MS`	2.2582	4.6233	9.6445	19.0842	38.9676
`OMP It.MS`	0.4222	0.8832	1.8516	3.7149	7.9209
`OMP MS`	0.3993	0.4722	1.4010	1.7271	5.3037
`TBB MS`	0.2653	0.4225	1.1002	1.8444	4.5177
`TBB Sort`	0.2641	0.5895	1.1488	2.6189	5.4585

Table 3.1: Run-time in seconds on a 12 core machine. Best run-time for each length is highlighted in green.

Overall, TBB MS performs better than OMP MS. This seem to suggest that the task scheduler is more efficient than the tasking model of OpenMP as implemented in GCC 4.5. Also, TBB MS not does exhibit the change in performance depending on how many cores are active, even though it uses the same cutoff value as OMP MS. This suggests the issue with OpenMP performance stems from calling Parallel Merge, which is implemented in TBB. Figure 3.6 confirms this by plotting results from the two recursive algorithms with Parallel Merge disabled. The performance of both TBB MS and OMP MS decreases when not using Parallel Merge, but OMP MS no longer displays the "jagginess" observed earlier.

3.4.2 Comparison of OpenMP and TBB

TBB MS offered the greatest speedup of all algorithms. While the complexity of implementing the parallel recursive algorithms was about the same, the TBB implementation requires more supporting code. This results in a longer development time.

The parallel merge algorithm was only implemented using TBB, since implementing the algorithm using OpenMP would be much more complex. OpenMP does not offer any mechanisms for separating the assignment of data and the computation itself. This flexibility along with the fact that TBB yielded greater performance speaks heavily in TBBs favor.

On the other hand, the iterative algorithm was only made parallel using OpenMP. While not performing nearly as well as the recursive algorithms, it can be argued that the speedup achieved is impressive when considering the effort put into it.

We conclude this chapter by noting that TBB offered better performance and flexibility while OpenMP provided an easier route to a parallel program. Which of these factors are more important varies depending on the task at hand.

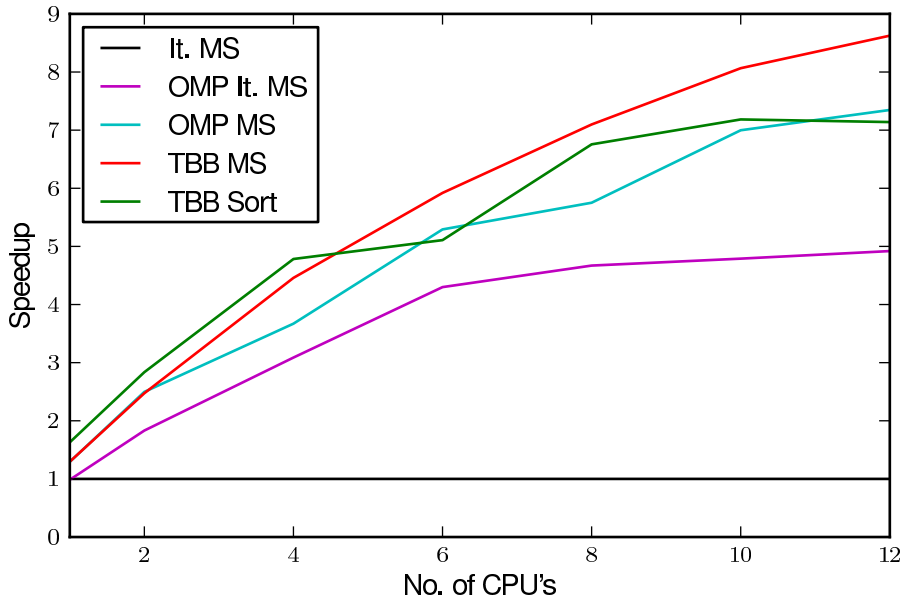

Figure 3.2: The speedup of each algorithm compared to the sequential algorithm as more cores are added. Input length is 2^{28}.

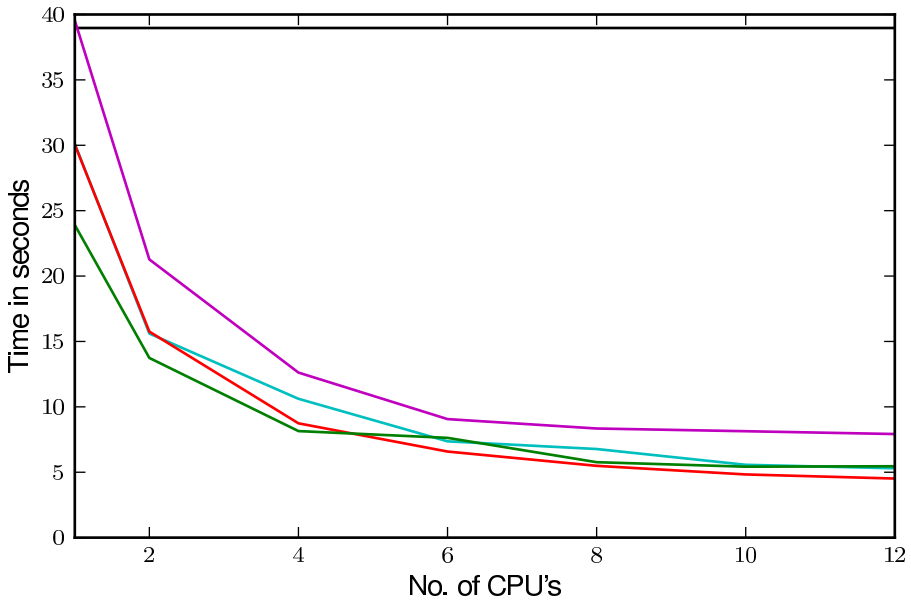

Figure 3.3: Same as Figure 3.2, but with running time in seconds.

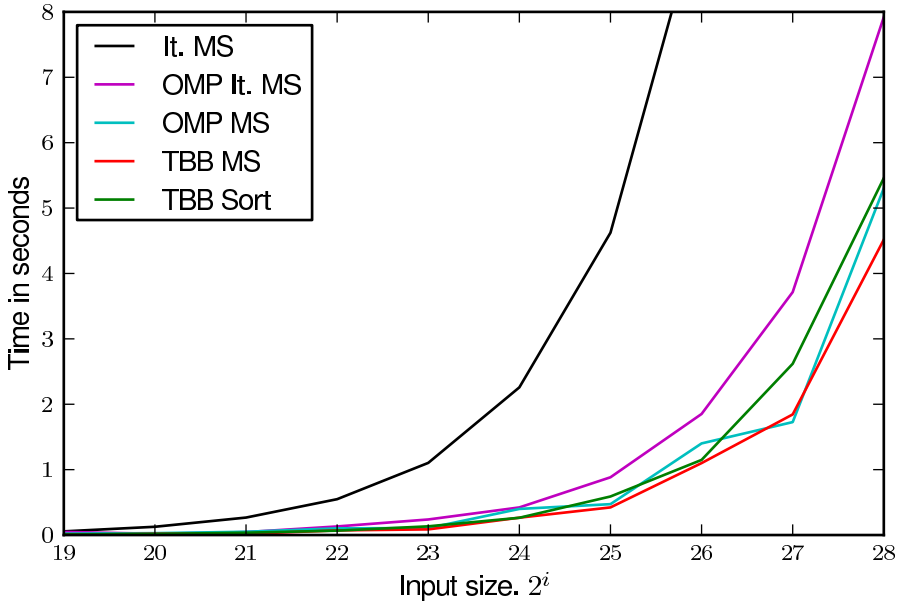

Figure 3.4: Running time in seconds on a 12 core machine with increasing input size.

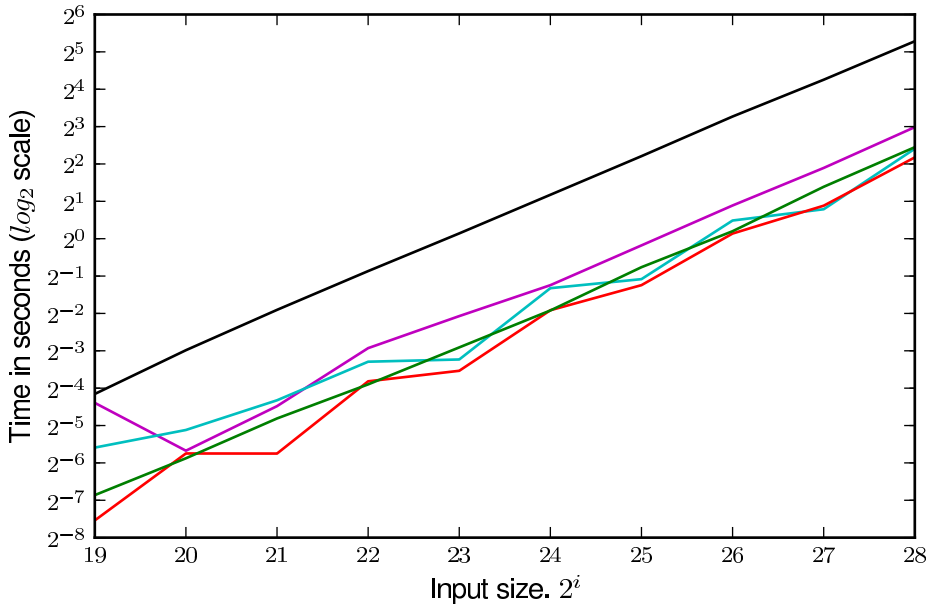

Figure 3.5: Same as Figure 3.4 but with a log_2 based y-axis.

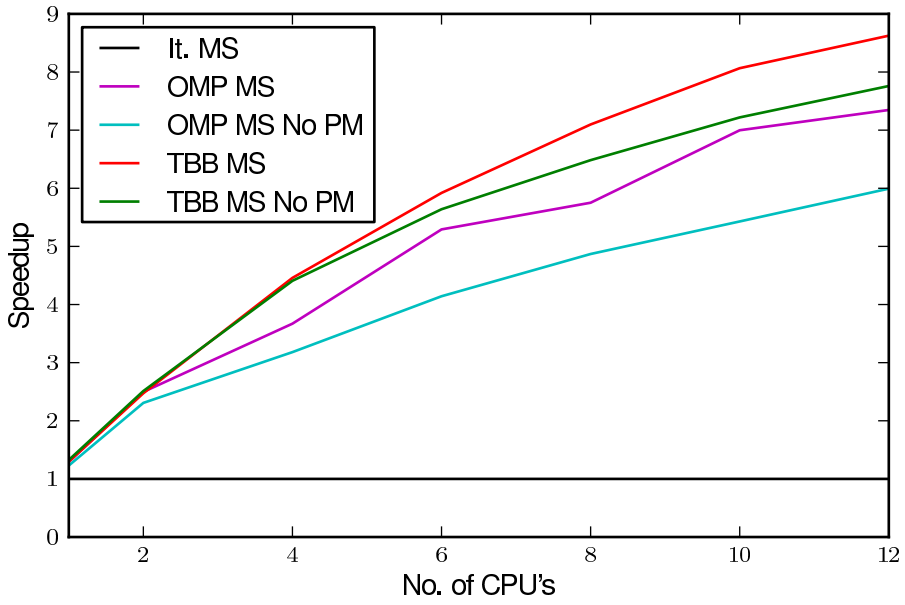

Figure 3.6: Comparison with and without Parallel merge on input of length 2^{28}

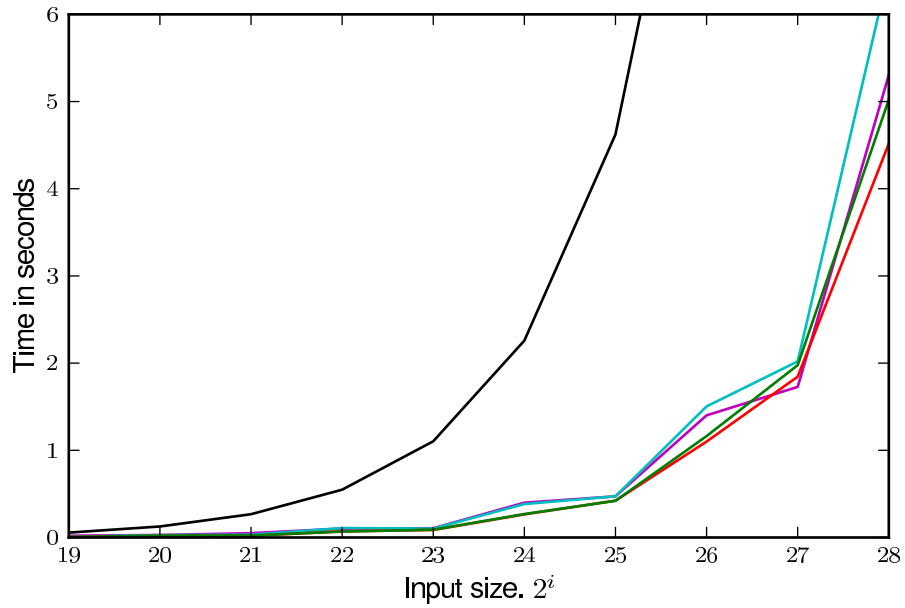

Figure 3.7: Same algorithms as Figure 3.6 but with running time in seconds as input size increases.

4

Parallel union-find for finding connected components in graphs

This chapter covers an experimental evaluation of several new implementations of disjoint set data structures for shared-memory systems. Algorithms that operate on these data structures are often referred to as Union-Find algorithms. These algorithms have several practical applications, among others in image decompositions and sparse matrix computations. Union-Find algorithms are also useful for finding minimum spanning trees and connected components in graphs. The latter of which will be the focus of this chapter. There has been previous effort to construct parallel Union-Find algorithms for shared memory systems. In [11], Cybenko *et al.* present a parallel Union-Find algorithm for both shared- and distributed memory systems. However, the results were not promising, showing a decrease in performance as more processors were added. In [2], Anderson and Woll describe an approach using *wait-free* objecs, but no experimental results were presented.

In [20], Patwary, Blair and Manne describes an experimental evaluation of sequential Union-Find algorithms for disjoint sets. The paper concluded that a simple Union-Find algorithm developed by Rem, ran faster than classic algorithms, despite having a higher asymptotic upper bound. This chapter builds on that work and presents an experimental evaluation of parallel variants of algorithms the authors described, implemented using OpenMP and TBB.

4.1 Overview

This chapter covers two set of algorithms that operate on disjoint set data structures, the Union-Find algorithm and Rems algorithm. Based on these two, several variants will be implemented. The following subsection formally present the algorithms.

4.1.1 Union-Find

The Union-Find algorithm operates on data structures known as disjoint sets. These are collections of disjoint sets $\{S_1, S_2, ..., S_n\}$ each containing elements from a finite universe U. Each set is represented by some element x, usually a member of the set. Two sets S_1 and S_2 are said to be disjoint if $S_1 \cap S_2 = \emptyset$. Each set is usually implemented as a rooted tree where the root node serve as the representative. Each node in a set contains a pointer to its parent and an *id*. Each component in a graph is represented by a set in the data structure. Initially, each node in a graph is represented as a singleton set. This is done with the MAKESET operation. If two nodes, x and y, share an edge, they belong to the same component. This is realized by setting the parent pointer of one node to point to the other. However, this linking has to be done between the root node of the components x and y belong to. The root node of the component containing x is denoted $r(x)$. The root is located with the FIND operation. It does this by traversing the parent pointers from x to $r(x)$. This path will be referred to as the *find-path*. A root node has the property $p(x) = x$. Upon finding the root nodes, the components containing x and y are linked with the UNION operation provided the components are not the same. When the algorithm terminates, each tree of parent pointers will represent a connected component. This is shown in Algorithm 2. Notice that the edges which result in a UNION operation together form a minimum spanning tree. In [30], Tarjan showed that UnionFind has time complexit $O(n + m\,\alpha(m, n))$ for any combination of m MAKESET, UNION and FIND operations on n elements, where α is the very slowly growing inverse of Ackermann's function.

Algorithm 2 Sequential Union-Find algorithm

```
1:  for x ∈ Nodes do
2:      MAKESET(x)
3:  end for
4:  for (u, v) ∈ Edges do
5:      ur = FIND(u)
6:      vr = FIND(v)
7:      if ur ≠ vr then
8:          UNION(ur, vr)
9:      end if
10: end for
```

As mentioned, FIND(x) returns $r(x)$; the root of the component x belongs to. This is done by traversing the find-path of x. By using compression techniques, which hopefully reduce the distance from x to $r(x)$, subsequent FIND operations for nodes on the same find-path become faster. Two such techniques are evaluated in this thesis.

The first, *Path Compression* (PC), uses a two-pass approach. After finding the root node a second pass sets the parent pointer of all nodes on the find-path to the root.

The second, *Path Splitting* (PS), sets the parent pointer of each node on the find-path to point to its grandfather. This technique has the advantage of only needing a single pass, while PC offers greater compression.

UNION(x, y) links the sets containing x and y. This is done by changing the parent pointer of one root node to point to the other. How the two roots are linked together will be referred to as our union-strategy. Two classic union-strategies will be evaluated.

The first, known as LINK-BY-RANK (LR) associates a rank value with each node. The rank value is initially set to 0. When linking two nodes with equal rank, the parent pointer of the node with the lowest rank is set to point to the node with greater rank value. When linking two root nodes with equal rank, the rank of the new root is incremented by 1. In all other cases the ranks stays the same. This implies that once a node is no longer root, its rank does not change. The rank of a parent is always greater than its children. This is known as the *increasing rank property*.

The other union-strategy evaluated is known as LINK-BY-INDEX (LI). This strategy links root nodes by comparing their id. Whichever one has the higher id becomes the new root. Linking of nodes with equal id cannot occur since the ids are unique. With this strategy, the id of a parent is always greater than the id of its children. This is known as the *increasing id property*. Whenever a point is made that holds true for both properties, they will together be referred to as the *increasing value property*.

With LR the length of a find-path is at most $\log(n)$ given a graph with n nodes. On the other hand, when using LI the length can be as much as n. One advantage of LI is that it requires less storage since, depending on the underlying data structure, ids do not have to be stored explicitly.

4.1.2 Rems algorithm for union find

As explained in [20], Rems algorithm belongs to a set of union-find algorithms called *interleaved* algorithms. Instead of doing FIND(x), then FIND(y), both are started simultaneously by setting $r_x \leftarrow x$ and $r_y \leftarrow y$. Then, whichever of r_x and r_y has the lowest parent id is moved one step along its find-path. This process continues in a loop until one of two conditions are met. If the nodes are in the same component,

then at one point $p(r_x) == p(r_y)$. In this case, the loop can terminate since no UNION is required. If this is not the case, then at some point one of the nodes will be a root node. Assume $r(r_x)$ has a lower id then $r(r_y)$. r_x is then linked with the component containing y as a sibling of r_y. This means that the two FIND operations do not necessarily have to traverse their entire respective find-paths.

Rem uses a compressing technique known as SPLICING, which works as follows. Assume r_x has a lower parent id than r_y. Before r_x gets assigned the parent value of x, the algorithm sets $p(r_x) = p(r_y)$. Since $p(r_y)$ must have a higher id than $p(r_x)$, this will hopefully result in a shorter find-path for x. Algorithm 3, shows the algorithm applied to an edge (x, y). SPLICING occurs on lines 7 and 12.

Algorithm 3 Rem with SPLICING

```
1:  r_x ← x, r_y ← y
2:  while p(r_x) ≠ p(r_y) do
3:    if p(r_x) < p(r_y) then
4:      if r(r_x) = r_x then
5:        p(r_x) = p(r_y), break
6:      end if
7:      z ← p(r_x), p(r_x) ← p(r_y), r_x ← z
8:    else
9:      if r(r_y) = r_y then
10:       p(r_y) = p(r_x), break
11:     end if
12:     z ← p(r_y), p(r_y) ← p(r_x), r_y ← z
13:   end if
14: end while
```

4.2 Analysis

This section presents how parallelism will be applied. We present two distinct approaches, one based on mutual-exclusion to prevent race-conditions and an alternative technique, which does away with mutual-exclusion in favor of a verification step performed after all edges have been processed.

After all nodes have been initialized, both the classic Union-Find algorithm and Rem process each edge one by one in a for loop. This can be made parallel by distributing the edges over available threads using a parallel for pattern. We then have to consider potential race-conditions that can arise in the parallel execution. A race-condition can occur whenever a thread writes to shared-memory. Here, this happens when the parent pointer of a node is updated and when the rank of a node is incremented. Parent pointer updates occur in UNION operations and also

FIND operations if compression is used. We start by considering only the UNION operation.

4.2.1 Union

If several UNION operations run in parallel, a race-condition can occur if UNION operations share one or more node among them. Consider two threads, t_1 and t_2, performing UNION(a, b) and UNION(a, c) in parallel, where a, b and c are root nodes. Assume a has the id of the three nodes. Since both t_1 and t_2 believe that a is a root node, both will attempt to update the nodes parent pointer. The pointer will end up being set to b or c, depending on which thread updates it last, resulting in two components. Now consider the sequential execution. If UNION(a, b) is performed first, a will no longer be a root node after the link has occurred. The algorithm would then proceed to find the representative of the components containing a and c. This would lead to UNION being called with b and c as parameters, making a, b and c part of the same component after the link. This means that updates to parent pointers need to occur in a mutual-exclusive code region. In addition, the UNION operation can not assume the nodes passed in are root nodes as the sequential operation can, since another thread might have updated the parent pointer. Next, we consider FIND operations.

4.2.2 Find

Observe that a FIND operation never updates the parent pointer of a root node. We now consider a FIND operation using PC. After finding the root node, another pass is done setting all nodes on the find path to point directly to the new found root. If another thread performs a FIND operation on a node residing on the same find-path, the same root node will still be found since the find-path cannot be broken. This because the parent pointer either points to the next node on the find-path or to the root node. This argument works for PS as well. The SPLICING compression used in Rem changes the parent pointer if the parent node of the sibling has a higher index value than the current node. Assuming that the race-condition that can occur in UNION is guarded by mutual-exclusion, SPLICING will not result in a broken find-path since both possible parent pointer settings will lead to the same root after linking. This means that as long as UNION operations are performed with mutual-exclusion, the parallel algorithms will yield the correct result.

4.2.3 Verification

As covered in Section 1.1, the maximum theoretical speed up of a parallel program is dependent on minimizing the sequential part. By removing locks, which represent sequential parts of the program, the potential speedup would increase. As stated,

this would result in a race-condition. We need to determine the consequences of race-conditions and show how these errors could be corrected.

Most importantly, we need to make sure that the increasing link value property is never broken. If this happens it might introduce a cycle in the data structure. This would lead to an infinite loop in the FIND operation for nodes on a find-path containing a cycle. When using LI, it is trivial to see that the increasing link value property cannot be broken. Since the id of a node never changes, race-conditions are irrelevant. A race-condition can, however, lead to a cycle when using LR. The same argument as applied in Subsection 4.2.1 can be used to show this, only applied to rank values instead of parent pointers. This means that only LI can be used as a union-strategy when doing verification.

The second issue that can arise as a result of a race-condition is that the wrong number of connected components get reported. Notice that, if this occurs, the algorithm can only report more components than there actually are, not less. This means that we can correct the answer by doing a second pass over all edges that resulted in a UNION operation, verifying that both endpoints of each edge are in the same component. The edges can be checked in parallel, but if any errors are found they cannot be fixed during this pass. Doing so would introduce the same race-condition we are trying to correct. Instead, edges where the endpoints are not in the same component are marked. After all edges have been processed, the marked edges can be passed as input to the Union-Find algorithm for further processing. We can then alternate between a Union-Find step and verification step until no edge given as input to the former result in a UNION operation. While this would yield an entirely parallel algorithm, for practical purposes and since the likely hood that an error will occur is quite low, a sequential step can be performed after the parallel verification step to fix any errors found.

The big downside to using verification instead of mutual-exclusion is that a minimum spanning tree is no longer defined by the edges which resulted in a UNION operation. There exists no easy way to correct this using a post-processing step.

4.3 Expressing parallelism with OpenMP and TBB

Aside from the parallel constructs, the implementations using OpenMP and TBB are similar. Algorithm 4 shows the general pattern for how all UNION operations work by using LR as an example. Given two pointers to root nodes, the algorithm first checks if both nodes have equal rank. If not, the node with the lowest rank value changes its parent pointer to the node with higher rank value. If the ranks are equal, the node pointed to by r_y becomes the root in the new component its rank is incremented by one. Note that Algorithm 4 is a sequential algorithm. For the algorithm to be thread-safe, some additional checking has to be performed. This was left out for clarity but is covered in Subsection 4.3.1.

Algorithm 4 LINK BY RANK

```
1: LinkByRank(r_x, r_y)
2: if rank(r_x > rank(r_y) then
3:    parent(r_y) = x
4: else if rank(r_x) < rank(r_y) then
5:    parent(r_x) = y
6: else
7:    parent(r_x) = y
8:    increment rank(r_y)
9: end if
```

Algorithm 5 shows pseudo-code for FIND with path-compression. On line 2, a copy of the input pointer is stored. After which the root node is found by traversing the find-path, updating the pointer until a root node is found. Remember that a root node is defined as a node with a parent pointer pointing to itself. On line 6, the second pass starts which does the compression. Notice that the condition for the while loop on line 6 uses the less than operator instead of checking for equality. The reason for this is that another thread might have updated the tree so the node currently pointed to by p_x is no longer root. If so, applying compression until a root node is found would break the increasing rank property.

Algorithm 5 FIND with path-compression

```
1: FindPC(p_x)
2: t ← p_x
3: while parent(p_x) ≠ p_x do
4:    p_x ← parent(p_x)
5: end while
6: while parent(t) < p_x do
7:    temp ← parent(t)
8:    parent(t) ← x
9:    t ← temp
10: end while
```

The algorithms were implemented using an object-oriented style. For the classic Union-Find algorithm, a top-level algorithm implements the pseudo-code given in Algorithm 2. When constructed, this object takes two objects as parameters which implement the UNION and FIND operation to be used. This allows all combinations of FIND and UNION techniques to be tested with minimal code duplication. In total, six UNION operations were implemented, three for each of the two union-strategies presented in Section 4.1.1. The three variations make use of locks provided by OpenMP, locks provided by TBB as well as a implementation without locking.

Three different `Find` methods were implemented, including both the compression techniques covered in Section 4.1.1 as well as a third which does not apply any compression. This will be referred to as *no compression*(NC). Section 4.2.2 concluded that all Find operations were thread-safe. Therefore the same implementations can be used with OpenMP and TBB.

For Rem, only one union-strategy and compression technique is used. Because of this the modular approach used for the classic Union-Find algorithm is not required. However, a mechanism was implement to toggle the use of SPLICING

Two variations of the verification step was implemented, one using OpenMP and one using TBB. This was done to avoid the performance hit associated with calling TBB code from OpenMP which was shown to be the case in Section 3.4.1.

Throughout the code examples, several typedefs will be used to minimize clutter. The types they refer to are shown in Listing 4.1. Listing every line of code would serve little purpose. Instead, we present only enough to cover all parallel constructs used in the implementation. Subsection 4.3.1 presents OpenMP and is divided into two parts. The first presents the top-level algorithm for Union-Find. This demonstrates the use of the parallel reduction directive. The second presents LI implemented with mutual-exclusion provided by OpenMP locks. Subsection 4.3.2

```
1  struct Node {
2    int parent, rank;
3  };
4  typedef std::pair<int, int>                NodePair;
5  typedef std::vector< Node >                NodeList;
6  typedef std::vector< NodePair >            EdgeList;
7  typedef tbb::concurrent_vector< NodePair > SafeEdgeList;
8  typedef tbb::spin_mutex                    MutexType;
9  typedef std::vector< MutexType >           MutexList;
10 typedef std::vector< omp_lock_t >          OmpLockList;
```

Listing 4.1: The typedefs used in the code examples

presents Rem with splicing implemented in TBB and explain the equivalent constructs for parallel reduce and locking.

4.3.1 OpenMP

Top-level Union-Find algorithm

Listing 4.2 shows the top level Union-Find algorithm. Since the algorithm is implemented as a class, it first needs to be instantiated. This is done by passing objects implementing FIND and UNION operations in addition to a char pointer and a boolean value conveying if this is a mutual-exclusion or verification instance. The char pointer is used for printing the name of the algorithm by the test harness and is not important for this discussion.

The algorithm is invoked by calling the objects `run` method with a graph instance given as a list of nodes and a list of edges, as shown on line 4. If locking is used the algorithm starts by initializing n locks, where n is the number of nodes in the graph instance. These locks will be used in the UNION operation. The algorithm then declares a vector which is used to store edges which resulted in a UNION operation. Notice that these edges are only store if they are needed for the verification step.

Parallelism is achieved through a parallel for directive appended with a reduction clause. A reduction clause takes as parameter one or more variables and a reduction operation to be performed at the end of the parallel region. Thread private copies of each variables are then made for each variable. In this case, the addition operation is performed over private copies of the variable `unions`. This variable is used assert that all algorithms perform the same number of UNION operations.

If this algorithm instance uses verification, the edges resulting in a UNION operation are added to a vector. Since a vector is not thread-safe, insertions must be performed in a critical section. A critical section is an OpenMP directive that defines a mutual-exclusive region. The critical section takes an optional name argument, in this case `add_union_edge`, to distinguish it from other critical sections. If omitted, all critical sections are treated as the same section. If a thread is currently executing the critical section and another thread attempts to enter it, it will block until the first thread exits the critical section. This is generally faster than a lock, but has some limitations. Most importantly only *code regions* can be guarded by a critical section. This can impact performance compared to a lock, since a lock can guard specific data elements updated in the region. Another limitation of critical sections is that they only have a single point of exit. This means that a thread cannot exit a critical section through a return, goto, continue or break statement.

After the parallel region defined by the `parallel for` directive terminates, a Verifier object is instantiated as necessary and the solution is verified. The verifier uses the same FIND and UNION operations as its caller.

Link By Index

Both union-strategies make use of the same OpenMP directives, so only LI is presented. Listing 4.3 shows the implementation. The method checks if the nodes

```
1  extern OmpLockList ompLocks;
2  UnionFindOmp::UnionFindOmp(Union& u_, Find& f_, const char* n, bool useLocking
       ):
3    UnionFind(u_, f_, n), lock(useLocking){}
4  int UnionFindOmp::run(NodeList& nodes, EdgeList& edges)
5  {
6    if(lock){
7      ompLocks.resize( nodes.size() );
8      for(size_t i = 0; i < ompLocks.size(); ++i)
9        omp_init_lock(&ompLocks[i]);
10   }
11   std::vector<NodePair> union_edges;
12   int unions = 0;
13   int size = (int) edges.size();
14 #pragma omp parallel for reduction(+:unions)
15   for(int i = 0; i < size; ++i) {
16     int xp = f.find(edges[i].first, nodes);
17     int yp = f.find(edges[i].second, nodes);
18
19     if(xp != yp)
20     {
21       int unionPerformed = u.doUnion(xp, yp, nodes);
22       unions += unionPerformed;
23       if(!lock && unionPerformed > 0) {
24 #pragma omp critical (add_union_edge)
25         {
26           union_edges.push_back(edges[i]);
27         }
28       }
29     }
30   }
31   if(!lock) {
32     Verifier v(nodes, union_edges, typeid(u).name() == linkbyrankid);
33     v.verifySolution();
34   }
35   return unions;
36 }
```

Listing 4.2: OpenMP parallel union find

passed as input are in fact root nodes. This is necessary since another thread might have updated the parent pointer of the input nodes. If the nodes are still root nodes, the thread attempts to acquire the lock associated with the lock. By requesting locks according to a predetermined ordering, deadlocks are avoided. If a thread is not able to acquire a lock, it blocks until the lock is released. If several threads block waiting for the same lock to be released, which of them get the thread is indeterministic. In other words, the lock is not fair.

Upon acquiring the lock the root check is performed again. This is done because another thread might have changed the parent pointers. This "double-checking" is a common idiom when locks are acquired given a condition being true. If the input nodes are still root nodes, the two trees represented by x and y are linked, after

which the locks have to be manually released. If the input nodes are no longer root nodes, the new roots need to be found. Some other thread might have done a UNION operation resulting in the two nodes belonging to the same component. If this is the case, 0 is returned, signaling that no UNION operation was performed by this call. If the nodes still belong to different components, the `doUnion` method is called recursively.

```
1  extern OmpLockList ompLocks;
2  int LinkByIndexOmpLock::doUnion(int x, int y, NodeList& nodes)
3  {
4    bool changed = false;
5    if( parentsNotUpdated(x, y, nodes) )
6    {
7      omp_set_lock(&ompLocks[std::min(x,y)]);
8      omp_set_lock(&ompLocks[std::max(x,y)]);
9      if( parentsNotUpdated(x, y, nodes) )
10     {
11       if ( x > y )
12         nodes[y].parent = x;
13       else
14         nodes[x].parent = y;
15     }
16     else changed = true;
17     omp_unset_lock(&ompLocks[x]);
18     omp_unset_lock(&ompLocks[y]);
19   }
20   else changed = true;
21
22   if(changed) {
23     NodePair np = findNewRoots(x, y, nodes);
24     if(np.first == np.second)
25       return 0;
26     return doUnion(np.first, np.second, nodes);
27   }
28   return 1;
29 }
```

Listing 4.3: LINK-BY-INDEX using OpenMP locks

```
1  RemTbb( const char* n, bool useSplicing, bool useLocking) :
2    Rem(n, useSplicing), lock(useLocking) { }
3
4  int run(NodeList& nodes_, EdgeList& edges_) {
5    SafeEdgeList union_edges;
6    RemTbbTask rtt(nodes_, edges_, union_edges, sp, lock);
7    if(lock)
8      mutexes.resize(nodes_.size());
9    parallel_reduce(blocked_range<size_t>(0, edges_.size(), 100000), rtt);
10   if(!lock) {
11     TbbVerifier::verifySolution(nodes_, union_edges, false);
12   }
13   return rtt.unions;
14 }
```

Listing 4.4: Top-level TBB Rem

4.3.2 TBB

Top-level Rem

We now turn our attention to how the code was made parallel using TBB. Listing 4.4 shows the object which implements the top-level Rem algorithm. To store the union edges, a concurrent vector is used. As mentioned in Section 2.1, this is a thread-safe container with the same semantics as an STL vector. A task body of type `RemTbbTask` is then instantiated and passed to a `parallel_reduce` parallel algorithm template, along with a `blocked_range`. This template requires the task body to implement a `join` method in addition to the overloaded "()" operator. This method is used to implement the reduction. Listing 4.5 shows the join method and the operator overload for the task body. When two tasks have completed, the task scheduler combines their results by invoking the join method of one task with a reference to the other as parameter. Since `join` is a regular C++ method, a TBB reduction is very flexible. As opposed to the OpenMP reduction clause, where only arithmetic operations can be performed to combine data, arbitrary computations can be performed. In this case, the only thing needed is to combine the respective tasks `unions` value. When all tasks have been completed, the task allocated on line 6 of Listing 4.4 will contain the accumulated value.

Rem task body

The computation performed by the task body consists of calling one of two Rem methods for each edge in its range. The choice of which depends on if this instance uses mutual-exclusion or verification. Listing 4.6 shows the implementation `doRemWithLock`. Only half the algorithm is listed, since the other half is a practically identical `else` block for the case where $p(x) \geq p(y)$. The method implements the algorithm shown in Algorithm 3 with some additional code to support locking,

```
1  void join(const RemTbbTask& x) { unions += x.unions; }
2
3   void operator()(const blocked_range<size_t>& r)  {
4     if(lock) {
5       for(size_t i = r.begin(); i < r.end(); ++i) {
6         unions += doRemWithLock(edges[i].first, edges[i].second, nodes);
7       }
8     }
9     else {
10      for(size_t i = r.begin(); i < r.end(); ++i) {
11        int u = doRem(edges[i].first, edges[i].second, nodes);
12        if(u > 0) {
13          unions += u;
14          union_edges.push_back(edges[i]);
15        }
16      }
17    }
```

Listing 4.5: Join method and overloaded "()" operator for `RemTbbTask`.

as well as the possibility to disable Splicing. On line 4, a check is done to see if $p(x) < p(y)$. If this is the case, another check is performed to see if x is a root node. As explained in Subsection 4.3.1, this is done since another thread might have performed a Union on this node. If the node is still root, the thread attempts to acquire a lock. When using Rem, only the node with the lowest parent id has to be locked, since the other node does not need to be a root node. The lock is created by passing a mutex to its constructor. If a thread attempts to acquire a lock held by another thread, the thread blocks while waiting. Like the OpenMP lock discussed in Subsection 4.3.1, the lock is not fair.

```
1  int RemTbbTask::doRemWithLock(int x, int y, NodeList& nodes) const {
2    while ( nodes[x].parent != nodes[y].parent) {
3      if ( nodes[x].parent < nodes[y].parent ){
4        if( nodes[x].parent == x)      {
5          MutexType::scoped_lock xlock(mutexes[x]);
6          if( nodes[x].parent == x) {
7            nodes[x].parent = nodes[y].parent;
8            return 1;
9          }
10         continue;
11       }
12       if(sp){
13         int z = nodes[x].parent;
14         nodes[x].parent = nodes[y].parent;
15         x = z;
16       }
17       else x = nodes[x].parent;
18     }
19    // Same code as above but with p(x) >= p(y)
20   }
21   return 0;
22 }
```

Listing 4.6: Rem algorithm with mutual-exclusion provided by TBB locks.

4.4 Test setup and Results

The experiments were conducted on a machine with 2 six-core AMD Operton 2431 processors (2.4 GHz) with 8 GB of memory. The machine was running Linux with kernel version 2.6.18. The program was compiled with GCC version 4.5.2 using the O3 optimization level. Version `tbb30_20100406oss` of the TBB library was used.

Since the performance of graph algorithms can vary depending on the topography of the graph, the algorithms were tested with several graph instances. The graphs used are shown in Table 4.1. The first graph in the table, *er*, is a random graph generated by GTgraph [6]. The second, lg, is a one-component graph were every node has a degree of one. This graph is included as a worst-case instance for verification algorithms. The three next graphs are randomly generated instances of small-world graphs. A small-world graph is a graph where most nodes are not neighbors, but most nodes can be reached from any other in just a few steps. Lastly, two real-world graphs were used. The first of which is used in linear programming and the second in medical science [13].

Test were performed on all algorithms for all graphs with an increasing number of cores utilized. At each number of cores, each algorithm ran 5 iterations. The running times presented are the average of the five iterations.

Graph	|V|	|E|	Comp
`er`	4,000,000	10,000,000	27385
`lg`	400,000	399,999	1
`sw1`	75,000	12233404	9465
`sw2`	100,000	16383918	34465
`sw3`	175,000	27339040	43929
`rw1` (af_shell10)	1,508,065	25,582,130	1
`rw2` (boneS10)	914,898	27,276,762	1

Table 4.1: The graph instances used in the experiments

Algorithm	er	lg	sw1	sw2	sw3	rw1	rw2
NC LR	2.0342	0.0427	0.2612	0.3814	0.7896	0.3905	0.4219
PS LI	1.5852	0.0406	0.3164	0.4603	0.9168	0.5574	0.6131
PS LR	1.5541	0.0437	0.3087	0.4511	0.8942	0.4571	0.4937
PC LI	1.5557	0.0404	0.2791	0.4052	0.8302	0.4585	0.5236
PC LR	2.0067	0.0442	0.2689	0.3943	0.8081	0.4037	0.4397
Rem SP	1.1249	0.0190	0.1118	0.1620	0.3625	0.2709	0.2563

Table 4.2: Run time in seconds for the sequential algorithms. The fastest algorithm for each graph is highlighted in green.

4.4.1 Results

Two algorithms were omitted from the experiment because they ran several orders of magnitude slower than the rest. The algorithms omitted are Rem without Splicing as well as classic Union-Find using NC and LI. In addition, NC was omitted from verification algorithms. The results are divided into three parts. First we present the running time of the sequential algorithms. Then we cover the parallel variants of the classic Union-Find algorithm. The third section presents the parallel Rem algorithms.

Sequential algorithms

Table 4.2 shows the running time for each sequential algorithm on each graph instance. The fastest algorithm by far for all graph instances is Rem. This confirms the findings presented in [20]. Of the classic Union-Find algorithms, none stand out as the fastest, with the ordering changing between graph instances. NC LR performs well on all graphs with the exception of `er`; a very sparse graph.

Parallel Classic Union-Find

We now discuss the variations of the classic Union-Find algorithm. Both the approaches using mutual-exclusion and verification will be covered. Table 4.3 shows the running times in seconds for each algorithm when utilizing 12 cores. The algorithm with the lowest running time is highlighted in green. The algorithm with the best run time of all is highlighted in red. As was the case with the sequential algorithms, no one combination of UNION and FIND operations stand out performing consistently better than the rest. Combinations using PS do however stand out as being the slowest of the three FIND operations overall.

When considering the algorithms using verification, `TBB PC LI` performed best on all graph instances. From this observation we can conclude that the combination of PC with LI outperforms PS with LI. Notice the extremely high run times for verification based OpenMP algorithms. The reason for this is that the algorithm adds edges which resulted in a UNION operation to a vector inside a critical section. This creates a region of code with high contention. This is particularly obvious for graph instance where a high percentage of edges result in a UNION operation. An example of this can be seen in the running time for `er`, where OpenMP verification algorithms are an order of magnitude slower than the TBB variants. This also shows that `concurrent_vector` is implemented efficiently.

Figure 4.1 shows the speedup of all classic Union-Find algorithms on `rw1`. The speedup is calculated from the fastest sequential non-Rem algorithm, which for this graph instance was NC LR. We consider the OpenMP implementations first. With the exception of the two verification algorithms, all algorithms manage to utilize additional cores efficiently. The three algorithms that offer the best speedup all use LR as union-strategy. Of the lock based algorithms, PS LI yields the least amount of speedup. We now consider the TBB implementations. Again, the three best algorithms use LR. However, the TBB algorithms are only able to achieve close to a 7x speedup factor, while the same algorithms in OpenMP pass an 8x speedup. The verification algorithms, while offering some improvement over the sequential algorithm, performs poorly. This means that the savings achieved by not using locks do not justify the second pass in this graph instance.

Figure 4.2 shows the results for a different graph; the largest small-world graph, `sw3`. Again, we start by considering the OpenMP implementations. For this graph instance, the amount of speedup is significantly less than for `rw1`. From 8 cores and upward, adding more cores has a negative effect on the performance of the algorithm. On this graph instance, the TBB implementations offer a better speedup, with the fastest of the locking algorithms being NC LR on 10 cores. From 10 cores onward though, any additional cores negatively impacts the performance. The algorithm with the best performance on this graph is a verification algorithm, PC LI. The algorithm continues to show speedup as more cores are added. The reason for this algorithm performing so well is because of the relative few number of edges which

result in a UNION operation. Since this graph instance is relative dense, the number of threads blocking while waiting for locks increase. Which shows by the dip in performance.

Based on `sw3`, it is likely locks are TBB locks are more efficient.

In total 840 iterations where performed using verification algorithms. Of these, an error caused by a race-condition occurred twice, both times in `er`. Both happened in PC LI when using TBB, once when 10 cores were used and once with 12. There is no reason to think that this means TBB is more prone to errors then OpenMP. We note however, that errors were extremely rare in these experiments.

Parallel Rem

As shown in Table 4.3, Rems algorithm with SPLICING was superior to all other algorithms tested in all but one graph, `lg`. However, the difference in execution time between PC LI and Rem implemented with OpenMP and mutual-exclusion was negligible. On all instances tested, locking was superior to verification. The difference between using OpenMP and TBB in algorithms were also minimal.

Figure 4.3 shows the speedup of the parallel Rem algorithms on `rw1`. Like we observed in Figure 4.1, verification algorithms perform badly on this graph instance. Both lock based algorithms show speedup until 10 cores are utilized. At which point the OpenMP implementation stagnates. The reason for this is unknowns. The TBB algorithm does however, continue to speedup.

Looking at Figure 4.4, which shows the speedup on `sw3`, we see that Rems algorithm using locking to not suffer from bad scalability as the classic Union-Find algorithms do. This is likely because, with Rem, only a single lock needs to be acquired per UNION operation.

Overall, the results for the parallel Rem algorithms look promising, with `TBB RemSP` with locking showing speedup as more cores were added across all graph instances.

4.4.2 Comparison of OpenMP and TBB

We will now cover the biggest differences between the OpenMP and TBB constructs applied in this chapter. We start by discussing locking. OpenMP, being designed to work both C/C++ as well as Fortran, do not offer the same level of abstraction as TBB does. When using locks, the programmer first has to manually initialize the locks by calling the `omp_init_lock` function. TBB on the other hand, implicitly initialize the locks through a default constructor. Also, OpenMP locks have to be manually acquired and released. TBB, being a C++ library, can make full use of the features offered by the language. An example of this is the `scoped_lock`, which makes use of a C++ feature known as *Resource Acquisition Is Instantiation*. This means that the lock is acquired at the time of declaration. When the code

declaring the lock goes out of scope, the lock is automatically released through the lock objects destructor. This simplifies the code and removes the risk of introducing bugs by forgetting to release a lock. This could be implemented for OpenMP locks as well by wrapping the acquisition and releasing in a class, but is not provided by default.

Secondly, TBB offers more flexible reductions since it is invoked as a standard method. The OpenMP reduction clause only supports primitive types as reduction values and only primitive reduction operations when combining them. The behavior of the TBB reduction template can somewhat be mimicked in OpenMP. Instead of using the reduction clause, thread private copies of complex objects can be declared inside a parallel region. At the end of this region, the objects can then be joined in a critical section. This would however not offer the same performance as only a single thread could execute the critical section at a time.

Which of the two parallel approaches offer the best performance varies between graph instances. The results seem to suggest that the locks offered by TBB are superior to their OpenMP counterparts. They also seem to suggest that parallel for loops are faster in OpenMP. These hypothesis have not been tested enough to draw any conclusions.

Algorithm	er	lg	sw1	sw2	sw3	rw1	rw2
`Omp NC LR L`	2.9076	0.0163	0.1091	0.1163	0.2391	0.0477	0.0530
`Tbb NC LR L`	1.8699	0.0296	0.0705	0.0828	0.1702	0.0593	0.0564
`Omp PC LI L`	2.7141	0.0147	0.1036	0.1085	0.2267	0.0530	0.0560
`Tbb PC LI L`	1.6515	0.0302	0.0733	0.0860	0.1710	0.0651	0.0602
`Omp PC LR L`	2.9474	0.0158	0.1074	0.1153	0.2326	0.0478	0.0487
`Tbb PC LR L`	1.8588	0.0314	0.0722	0.0822	0.1683	0.0563	0.0544
`Omp PS LI L`	2.7679	0.0153	0.1912	0.2519	0.4075	0.0646	0.0628
`Tbb PS LI L`	1.6986	0.0306	0.1633	0.2164	0.3528	0.0778	0.0792
`Omp PS LR L`	2.9643	0.0164	0.1909	0.2432	0.3977	0.0514	0.0510
`Tbb PS LR L`	1.8991	0.0322	0.1553	0.2069	0.3235	0.0615	0.0579
`Omp PC LI V`	6.1788	0.5627	0.1178	0.1305	0.2638	2.1804	1.2751
`Tbb PC LI V`	0.6186	0.0734	0.0400	0.0520	0.1007	0.2197	0.1378
`Omp PS LI V`	6.1658	0.5607	0.2034	0.2665	0.4356	2.2203	1.2796
`Tbb PS LI V`	0.6702	0.0756	0.1310	0.1905	0.2954	0.2337	0.1488
`Omp RSP L`	0.2241	0.0149	0.0190	0.0267	0.0512	0.0500	0.0426
`Tbb RSP L`	0.1758	0.0185	0.0254	0.0322	0.0519	0.0417	0.0385
`Omp RSP V`	6.6109	0.5880	0.1090	0.1199	0.2413	2.2599	1.3307
`Tbb RSP V`	0.4909	0.0610	0.0230	0.0303	0.0558	0.2043	0.1052

Table 4.3: Execution time in seconds when utilizing 12 cores. The fastest algorithm for each related algorithm type is highlighted in green. The fastest overall is highlighted in red.

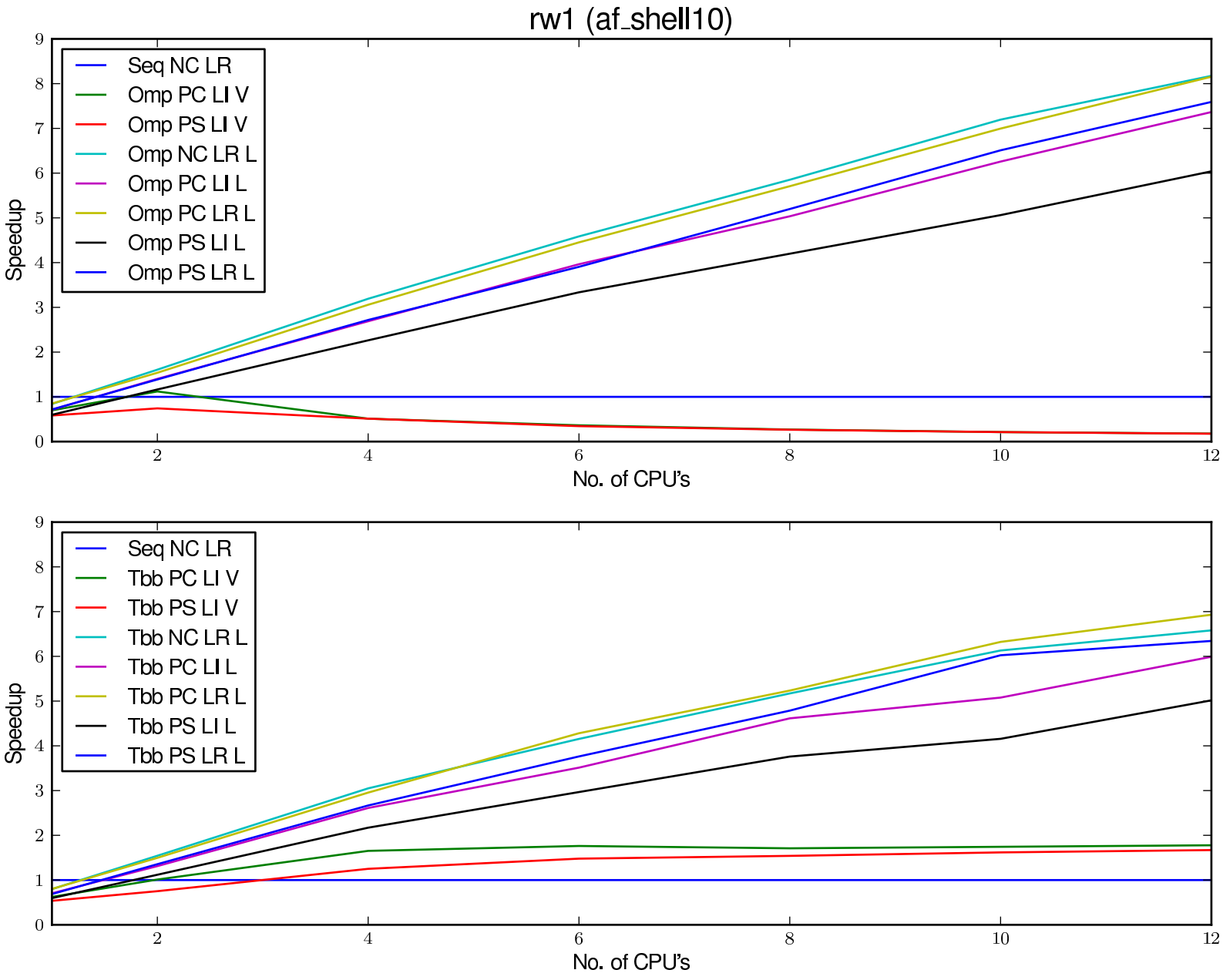

Figure 4.1: Speedup of all classic Union-Find variations compared to sequential NC LR on rw1

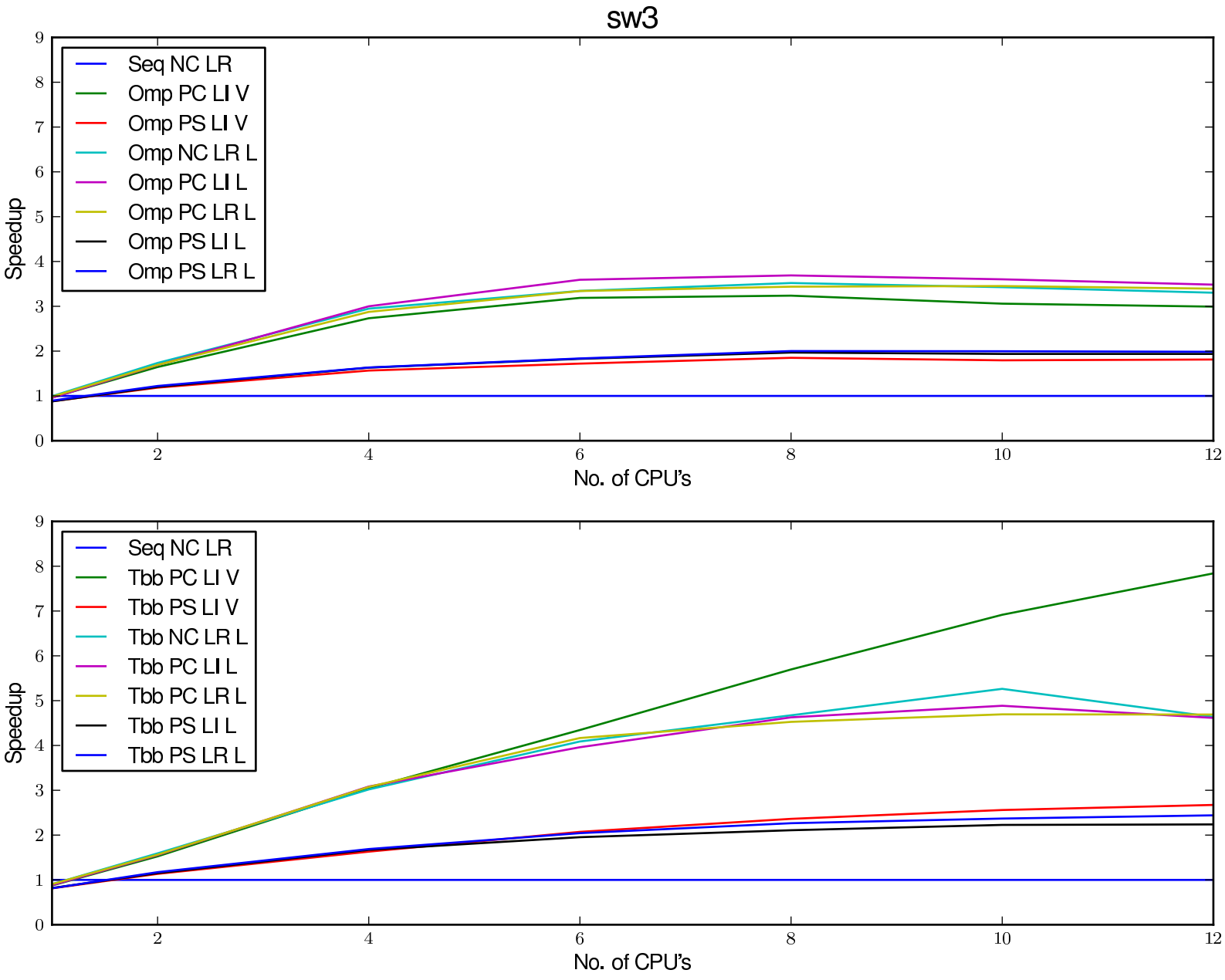

Figure 4.2: Speedup of all classic Union-Find variations compared to sequential NC LR on sw3

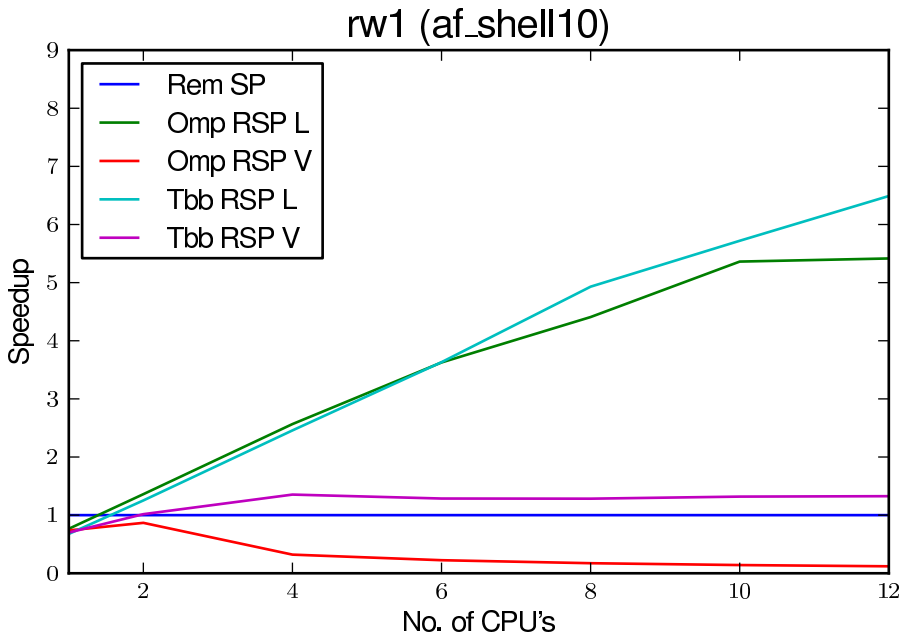

Figure 4.3: Speedup compared to sequential RemSP as more cores are added on rw1

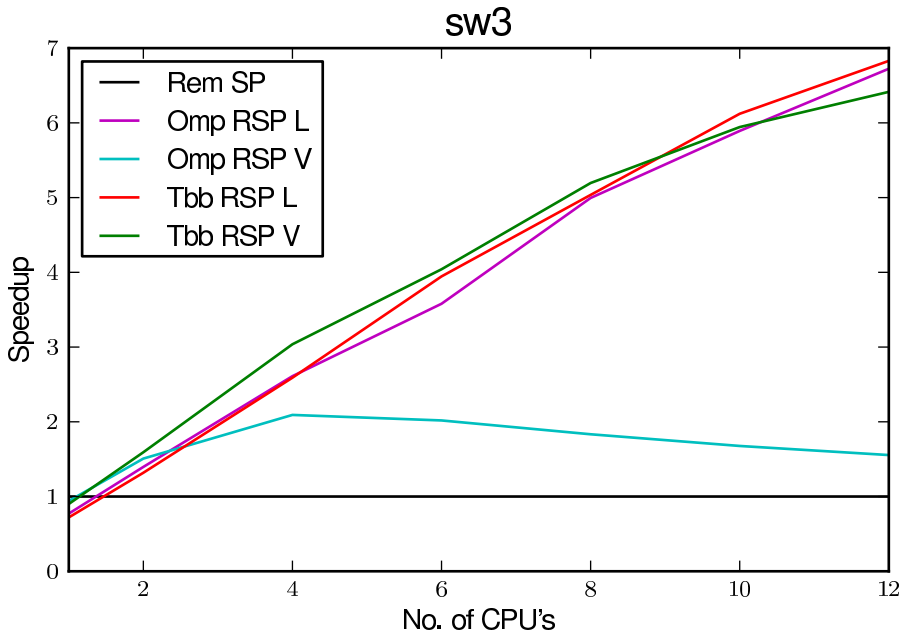

Figure 4.4: Speedup compared to sequential RemSP as more cores are added on sw3

5 Conclusion

In this thesis we have explored ways to parallelize code using OpenMP and TBB. In addition, we have presented a parallel algorithm for disjoint set data structures that shows promise with regards to scalability. This closing chapter offers an overview of possible further work and presents some final thoughts on OpenMP and TBB.

5.1 Further work

The parallel Rem implementation presented in Chapter 4 looks promising. The algorithm showed significant speedup in all experiments. These findings should be verified and tested more extensively, since they could have practical implications. The results offered in this thesis are not thorough enough to conclude that the algorithm performs well for all variations of graph. It would also be interesting to see experiments on a machine with more than 12 cores, since the tests never reached a point where adding more cores resulted in poorer performance.

Another possible project would be to develop an implementation of Rem with the goal of having it accepted into the Boost C++ library [14]. Currently, Boost only offers algorithms for disjoint sets based on the classic union-find algorithm.

5.2 Closing thoughts on OpenMP and TBB

As mentioned several times earlier, OpenMP and TBB shows promise in different situations. For basic loop parallelizations, OpenMP seems to have the edge over TBB because of its minimal syntax and intuitive semantics. This ease in expressing

basic parallelism does however come at the cost of flexibility. Past a certain complexity threshold, expressing parallelism with OpenMP becomes awkward. This point has not been shown to a great extent in the parallel algorithms presented here, with Parallel Merge being the exception 3.3.4. This since few advanced constructs were needed. OpenMP has however, come along way with the introduction of the tasking model in version 3.0. For the next version of the specification, several interesting features have been proposed [4]. These include better error handling, better interoperability with other threading packages, support for transactional memory and improvements to the tasking model. These features would go along way in bridging the gap between the two approaches. Additional features is not necessarily positive. By introducing more and more advanced constructs, the OpenMP specification does run the risk of loosing its focus on the areas where it shines.

TBB, being a library and consisting only of pure C++ code, makes it a potentially better fit in enterprise settings. In large projects spanning several years the ability to understand how different components of a system communicate becomes greater. The abstractions provided by TBB allows this to be expressed more clearly and at a higher level. Also, the building blocks provided by TBB makes it possible to express complex forms of parallelism. These advanced constructs have not been needed for the algorithms presented in this thesis,and as such any examples of these constructs would be contrived. The ability to express advanced forms of parallelism has increased further in recent months with the addition of a message passing system for shared memory systems to the library [18].

Performance wise, TBB showed a slight edge overall in the experiments conducted for this thesis, but not enough to conclude that it is superior to OpenMP. To conclude this it might be useful to conduct experiments with OpenMP implementations in C, since that would likely result in improved performance.

Overall, the choice of which approach to choose depends entirely in the problem at hand. Hopefully, this thesis has been able to give the reader an intuition to aid in this choice.

Bibliography

[1] G.M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, spring joint computer conference*, pages 483–485. ACM, 1967.

[2] R.J. Anderson and H. Woll. Wait-free parallel algorithms for the union-find problem. In *Proceedings of the twenty-third annual ACM symposium on Theory of computing*, pages 370–380. ACM, 1991.

[3] Diego Andrade, Basilio B. Fraguela, James Brodman, and David Padua. Task-parallel versus data-parallel library-based programming in multicore systems. *Parallel, Distributed, and Network-Based Processing, Euromicro Conference on*, 0:101–110, 2009.

[4] ARB. Openmp website. `http://www.openmp.org`.

[5] E. Ayguadé, A. Duran, J. Hoeflinger, F. Massaioli, and X. Teruel. An experimental evaluation of the new openmp tasking model. *Languages and Compilers for Parallel Computing*, pages 63–77, 2008.

[6] D.A. Bader and K. Madduri. GTGraph: A synthetic graph generator suite. *Atlanta, GA, February*, 2006.

[7] D. Bitton, D.J. DeWitt, D.K. Hsaio, and J. Menon. A taxonomy of parallel sorting. *ACM Computing Surveys (CSUR)*, 16(3):287–318, 1984.

[8] R.D. Blumofe, C.F. Joerg, B.C. Kuszmaul, C.E. Leiserson, K.H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 207–216. ACM, 1995.

[9] C++ Standard Committee. C++ standard committee website. `http://www.open-std.org/jtc1/sc22/wg21/`.

[10] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to algorithms*. The MIT press, 2009.

[11] G. Cybenko, TG Allen, and JE Polito. Practical parallel union-find algorithms for transitive closure and clustering. *International journal of parallel programming*, 17(5):403–423, 1988.

[12] L. Dagum and R. Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science & Engineering, IEEE*, 5(1):46–55, 2002.

[13] T. Davis. University of Florida sparse matrix collection. *NA Digest*, 97(23):7, 1997.

[14] B. Dawes and D. Abrahams. The boost library. `http://www.boost.org`.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, volume 206. Addison-wesley Reading, MA, 1995.

[16] D. Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[17] J.L. Gustafson. Reevaluating Amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[18] Intel. Threading building blocks open-source website. `http://www.threadingbuildingblocks.org`.

[19] Y.A. Liu and S.D. Stoller. From recursion to iteration. *ACM SIGPLAN Notices*, 34(11):73–82, 1999.

[20] F. Manne and M.M.A. Patwary. An experimental evaluation of union-find algorithms for the disjoint-set structure.

[21] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, 2004.

[22] D.R. Musser and A. Saini. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison Wesley Longman Publishing Co., Inc. Redwood City, CA, USA, 1995.

[23] OpenMP. The openmp specification. `http://openmp.org/wp/openmp-specifications/`.

[24] R. Rasala. Function objects, function templates, and passage by behavior in C++. In *Proceedings of the twenty-eighth SIGCSE technical symposium on Computer science education*, pages 35–38. ACM, 1997.

[25] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2007.

[26] A. Robison, M. Voss, and A. Kukanov. Optimization via reflection on work stealing in TBB. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8. IEEE, 2008.

[27] W. Shu and L.V. Kale. Chare kernel–a runtime support system for parallel computations. *Journal of Parallel and Distributed Computing*, 11(3):198–211, 1991.

[28] B. Stroustrup. A History of C++. *ACM SIGPLAN Notices*, 28(3):271–297, 1993.

[29] M. Süß and C. Leopold. A user's experience with parallel sorting and openmp. *Proceedings of the Sixth Workshop on OpenMP (EWOMP'04)*, 2004.

[30] R.E. Tarjan and J. Van Leeuwen. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2):245–281, 1984.

[31] Wikipedia. Wikipedia's tbb page. `http://en.wikipedia.org/wiki/TBB`.

[32] B. Wilkinson and M. Allen. *Parallel programming: techniques and applications using networked workstations and parallel computers*. Prentice Hall, 1998.

List of Figures

List of Tables

Listings