FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's thesis

Ramón Salvador Soto Rouco

Efficient algorithms and data structures on multi-core computers

Advisor: Jyrki Katajainen

Submitted: August 2010

Abstract

The increasing computation power in modern computers in the form of several cores per processor and more processors, makes it necessary to rethink or to redesign sequential algorithms and data structures. An obvious approach would be to use parallelism. Since there are several different programming models to implement parallelism, which one should be used?

In this thesis various parallel programming models are studied from a practical approach and evaluated with emphasis on efficiency on multi-core computers. A *multi-threaded* framework, based on a *shared-memory* model, is designed and implemented. The framework offers a *pure and simple C++ API*, a *limit* on usage of threads, *load-balance* between work performed by threads, *atomicity* when a shared data structure state is changed and *low memory* usage.

A *helping technique* is introduced. This technique can distribute work between inactive processors in *logarithmic time*. Also two types of barriers are introduced: a *global thread barrier*, which reduce the use of synchronization barriers; and a *job barrier*, which can execute the work binded to a barrier if no other thread is performing the task instead of just waiting.

Satisfactory results are presented when comparing the presented framework to other available frameworks and libraries. The comparison shows that similar practical efficiency on the parallel implementation of *sorting* algorithms is obtained, sometimes even better, with use of less memory.

Acknowledgements

I thank my advisor Jyrki Katajainen for his help, guidance and support in the understanding and writing of this thesis.

Furthermore I thank Martin Zachariasen, David Pisinger, Marco Danelutto, Antonio Cisternino and the Departments of Computer Science at Universities of Copenhagen and Pisa for making resources available so I could execute performance test on different computer architectures.

Finally I thank Joakim Ahnfelt-Rønne and Jørgen Haahr for proofreading the report and Alexander Færøy for proofreading the code.

Contents

1 Introduction **1**

1.1 Motivation 1

1.2 Multi-core computers 2

1.3 Parallel algorithms and data structures 4

1.4 An efficient library for C++ 4

2 Methodology **7**

2.1 Parallel random access machine 7

2.2 Algorithmics as Algorithm Engineering 9

2.3 Asymptotic performance measures 10

2.4 Practical efficiency measures 12

3 Existing frameworks and libraries **15**

3.1 POSIX threads (pthreads) 15

3.2 OpenMP 17

3.3 Cilk++ 19

3.4 C++CSP2 22

3.5 MCSTL 22

4 EAD++ framework **25**
4.1 Analysis 26
4.2 Design and implementation 27
4.3 API 32

5 Sorting in parallel **35**
5.1 Bitonicsort 36
5.2 Mergesort 42
5.3 Quicksort 45

6 Finding minimum spanning trees in parallel **49**
6.1 Prim (Dijkstra-Jarník-Prim, DJP) 49
6.2 Kruskal 52
6.3 Borůvka (Sollin) 57

7 Experimental setup **63**

8 Results and discussion **67**
8.1 Sorting 67
8.2 Finding minimum spanning trees 69
8.3 Discussion 71

9 Closure **73**
9.1 Conclusion 73
9.2 Future work 74
9.3 Framework availability 76

Bibliography **79**

A Source code **81**
A.1 Efficient library for C++ 81
A.2 EAD++ framework 81

B Test **83**

B.1 Correctness 83

B.2 Benchmark 83

C Performance measures for sorting algorithms **85**

C.1 On dual-hyperthreading server 85

C.2 On dual-quadcore server 94

C.3 On tetra-octacore server 107

Chapter 1

Introduction

1.1 Motivation

The increasing computation power in modern computers in form of several cores per processor and more processors, makes it necessary to rethink or to redesign sequential algorithms and data structures.

Consider a sequential algorithm that is executed in $T(n)$ time. If the task is fully parallelizable, the execution time of the algorithm would be $\frac{T(n)}{k}$ time when having k cores running in parallel.

Jordan Hubbard, Director of Engineering of Unix Technologies at Apple and co-founder of the FreeBSD project, announced at *Open Source Days 2008*[1] that Intel, according to their plan, has the intention to ship computers in 2015 with *ONE MILLION* cores. With that amount of cores, sequential algorithms and sequential data structures will not be able to utilize optimally the available hardware resources.

With all that computational power is it possible to design efficient algorithms and data structures on multi-core computers, and if so which framework should be used? An obvious approach would be to use parallelism. Looking at [24] an increased performance is obtained if parallelized algorithms from the *Multi-Core Standard Template Library (MCSTL)* are executed on a computer with a greater amount of cores. On the other hand we have

[1]http://www.usenix.org/event/lisa08/tech/hubbard_talk.pdf

[11, 9], both implementation of a *Software Transactional Memory library for C++ (ESTM and DikuSTM)*, where the performance decreases after adding seven or eight threads. There are several different programming models to implement parallelism, including:

Parallelizing algorithms: Take well known efficient sequential algorithms and redesign them so that the execution of non-overlapping operations can be done concurrently on several processors, reducing the execution time of the algorithm.

Software Transactional Memory: By introducing a transactional model similar to databases transactions, concurrent read and write can be done on a shared data structures ensuring the following properties: *atomicity*, *consistency* and *isolation* (ACI).

Skeletons [7]: Useful patterns of parallel computation and interaction can be packaged up as *frameworks* or *template constructs.*

Communicating sequential processes (CSP) [13]: Formal language for describing patterns of interaction with messages between processes in concurrent systems.

It is necessary to make an experimental study on the proposed models and evaluate them in combination with the release of an *open source* parallel framework that can be used by researchers in future experimental studies.

1.2 Multi-core computers

A *multi-core computer* is defined as a computer having at least one *central processing unit (CPU)* with several cores. The *core* is the part of the processor that performs the execution of the instructions. On a *single-core* processor only one instruction can be processed at a given time while *multi-core* processors can execute several instructions. This is interesting because it allows computers with only one CPU to be able to run parallel applications.

The naming convention for describing how many cores a *multi-core* processor has, is done by prefixing with a word for the number of cores: *dual-core, quad-core, octa-core*, etc.

Take two processors from *Intel*, *Intel Core 2 Duo Processor E8200 (6M Cache, 2.66 GHz, 1333 MHz FSB)*[2] and *Intel Core 2 Quad Processor Q9400S (6M Cache, 2.66 GHz, 1333 MHz FSB)*[3], with the same *cache*, *CPU clock speed*, *FSB Speed* and where the second processor has twice the amount of cores. Assuming that we have the fully parallelizable application from Section 1.1, the computation can be done in half the time.

A thing to have in mind when speaking about *multi-core* processors is that an instance of an executed application, a *process*, can create several independent streams of instructions that can be scheduled to run concurrently by the operating system, called *threads*. The ideal mapping between the *threads* and *cores* is usually *1-to-1* but it is not determined for all architectures. *Sun's UltraSPARC T1 octa-core*[4] processor can execute *32* threads concurrently, *4* on each core. *Intel's multi-core* processor with *Hyper-Threading Technology*[5], can execute two virtual threads on each core, assuming that not all of the instructions on a core are used in each clock cycle. The performance in the *Hyper-Threading Technology* depends on if the threads can be executed in the same clock cycle.

In literature [14, 8] the running time of an algorithm on a computer with p processors is defined as $T_p(n)$ or T_p respectively. Where [14] defines p as a processor, the terminology of multi-core was not defined when the book was written and [8] defines p as both processors and cores.

In this thesis a new terminology is introduced taking into consideration that a core can execute several threads concurrently:

$T_k(n)$: The running time of an algorithm on a computer with k processors, where k is defined as a bijection of the k processors to the optimal amount of threads that can be executed concurrently on a given *hardware processor.*

The new terminology will ensure that no unexpected results, such as a significant lower execution time, will occur when performing practical efficiency measures on algorithms.

[2]http://ark.intel.com/Product.aspx?id=33909
[3]http://ark.intel.com/Product.aspx?id=40814
[4]http://www.oracle.com/us/products/servers-storage/microelectronics/030990.htm
[5]http://www.intel.com/info/hyperthreading/

1.3 Parallel algorithms and data structures

The study will be based on less-know parallel algorithms as well as the parallelization of well-known sequential algorithms in literature [14, 27, 8] and implemented on different *C++ parallel frameworks*. The chosen *parallel model* is the *shared-memory model*, where *k processors* have access to a shared memory unit concurrently in order to exchange information. Other models such as the *network model or distributed model*, where several computers exchanging information concurrently over a network or specific models as *CUDA* for *NVIDIA* graphic cards will be excluded. Because of this limitation, techniques as *skeletons* which only have one C++ parallel library, *The Münster Skeleton Library (Muesli)* [6], will be excluded because it built on the *Message Passing Interface (MPI)*, a framework that *serializes* and sends data structures in messages in order to exchange information. It is most suited for sending serialized data over a network between computers. It is possible to use on a single machine, but the serialization overhead of the data transformation to communicate between core/processors would be excessive and unnecessary.

The *multi-threaded* frameworks based on the shared-memory model that will be reviewed are: *POSIX Threads*, *OpenMP*, *MCSTL*, *Cilk++*, *C++CSP2* and can be seen in Chapter 3.

1.4 An efficient library for C++

The library presented in this thesis offers efficient *sorting* and *minimum-spanning-tree* algorithms that have a lower execution time and at least the same *worst-case* asymptotic time complexity as the best sequential algorithm.

The two types of the algorithms were chosen based on that some sorting algorithms are suited for parallelization and do not depend on complex data structures. While *minimum-spanning-tree* algorithms are not trivially parallelizable and depend on complex data structures such as *sets* and *heaps*. The source code to the implemented algorithms can be seen in Appendix A.

The algorithms are implemented with the *Multi-Core Standard Template Library (MCSTL)*, *Cilk++* and the *Efficient Algorithm and Data Structures framework for C++ (EAD++)*, developed under this Master's thesis. The *EAD++* is a *multi-threaded* framework based on a *shared-memory* model and offers a *pure and simple C++ API*, a *limit* on usage of threads, *load-balance* between work performed by threads, *atomicity* when a shared data structure state is changed and *low memory* usage. A complete description can be seen in Chapter 4.

Pure C++ is defined by Bjarne Stroustrup in [25] as *"the language independently of any particular software development environment or foundation library (except the standard library, of course)"*. By offering a *pure and simple C++ API*, researches can implement and add algorithms to the library ensuring that the minimal requirements for *asymptotic performance measures* and *practical efficiency measures* are met. The criteria to measure the *asymptotic performance* and *practical efficiency* besides the methodology to design, analyze, implement and test algorithms can be seen in Chapter 2.

Satisfactory results are presented when comparing *EAD++* to other available frameworks and libraries. The comparison shows that similar practical efficiency is obtained, sometimes even better, with the use of less memory. In order to ensure that the presented results are correct a set of test are produced. Both the results and the test can be reproduced by the reader, for more information on results and correctness test look at Chapters 8 and 7 respectively.

Chapter 2

Methodology

In this chapter a definition of the different types of *parallel random access machine* models of computation used for the studied algorithm will be specified. Afterwards, the model used to *design*, *analyse*, *implement* and *test* parallel algorithms with emphasis on that they will be executed on different computer architectures, will be presented. Finally the criteria for accepting or rejecting the studied algorithms based on asymptotic *time* and *space* complexities, *asymptotic performance measures* and *practical efficiency measures* will be described.

2.1 Parallel random access machine

The *parallel random access machine (PRAM)* is an abstract theoretical computer model used to deal with the constraint where k processors are allowed to access the same memory address concurrently when designing algorithms for a shared memory system. A visual representation can be seen in Figure 2.1.

For each of the studied algorithms in this thesis, the type of *parallel random access machine* model of computation will be specified. The three models are:

Exclusive Read Exclusive Write (EREW): Different locations in the shared memory can be read by or written to exclusively by only one processor in the same clock cycle.

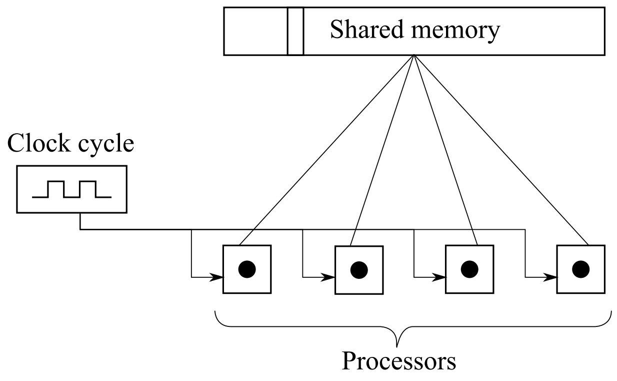


Figure 2.1: Representation of a parallel random access machine (PRAM) where *k processors* access a single shared memory cell in a clock cycle.

Concurrent Read Exclusive Write (CREW): Same locations in the memory can be read by several processors but only written to exclusively by one processor in the same clock cycle.

Concurrent Read Concurrent Write (CRCW): Same locations in the memory can be read by or written to by several processor in the same clock cycle. To this particular model, there are three submodels, based on the *write* constraint:

Priority: processors are assigned distinct priorities, where the one with the highest priority is allowed to perform the write to memory.

Arbitrary: A random processor is allowed to write.

Common: All processors are allowed to write to memory *iff* the values are equal. It is the algorithm's responsibility to ensure this condition holds.

An example could be to calculate the logical *OR* from n boolean values with k processors, where $k = n$. The calculation can be performed in *logarithmic* time, $O(\lg n)$, if the chosen model is either *EREW* or *CREW*. A binary recursive tree needs to be done and at each level where all the operations can be performed concurrently. If the chosen model is *common CRCW*, the calculation can be performed in *constant* time, $O(1)$. The initial value in the result slot is *0* and a processor will write the local value iff it equals *1*. It is important that algorithm do not write another number than *1* to the shared memory. If this is not the case and several processes write different values at the same time to the result slot corrupting the final result, then a *race condition* will arise.

Two things to have in mind when designing an algorithm based on these models is that *EREW* and *CREW* algorithms can be executed on a *CRCW* model but not the other way around. And the other is to prioritize correctness before efficiency.

2.2 Algorithmics as Algorithm Engineering

The chosen model to *design*, *analyse*, *implement*, *test* and measure the *asymptotic performance* and *practical efficiency* of the studied algorithm is *Algorithmics as Algorithm Engineering*[1]. The approach is based on an *eight step* model that is a combination of both a *theoretical* and *practical* model. As can be seen in the Figure 2.2 the *first* step is to define a *realistic model.* In this study the two underlying models, *application* and *machine*, to the *realistic model* will be defined as:

Application: several *C++ frameworks* that implement parallelism.

Machine: several *real computers* with different type and amount of processors and RAM. It is obvious to see here how the *simple computer model*, with one processor and infinite amount of RAM, from the *theoretical model* does not fit.

The next four steps: *design*, *analysis*, *implementation* and *experiments* are the core steps of the model. By using an *iterative developments model* [3], an algorithm is designed, analyzed and a *hypothesis* is presented. Afterwards the algorithm will be implemented, in this case in *C++*, and then initial experiments will be made. If the initial experiments show that the hypothesis does not hold, a new iteration can be made. If every time the *fifth* step is reached and the hypothesis is still not fulfilled, then the algorithm must be rejected. In case that the hypothesis holds, a new iteration to the *seventh* step will be made. In this step, experiments are made on *real data.* If the results are not satisfied a new iteration in the previous four steps can be done and if they are satisfied then the algorithm can be added to a library or to an application, in the *sixth* and *eighth* steps respectively.

Hypothesis will be presented on basis of asymptotic *time* and *space* complexities and *asymptotic performance measures* and *practical efficiency measures.* A minimal requirement for the asymptotic time complexity is that

[1]http://algo2.iti.kit.edu/sanders/courses/bergen/bergen2.pdf

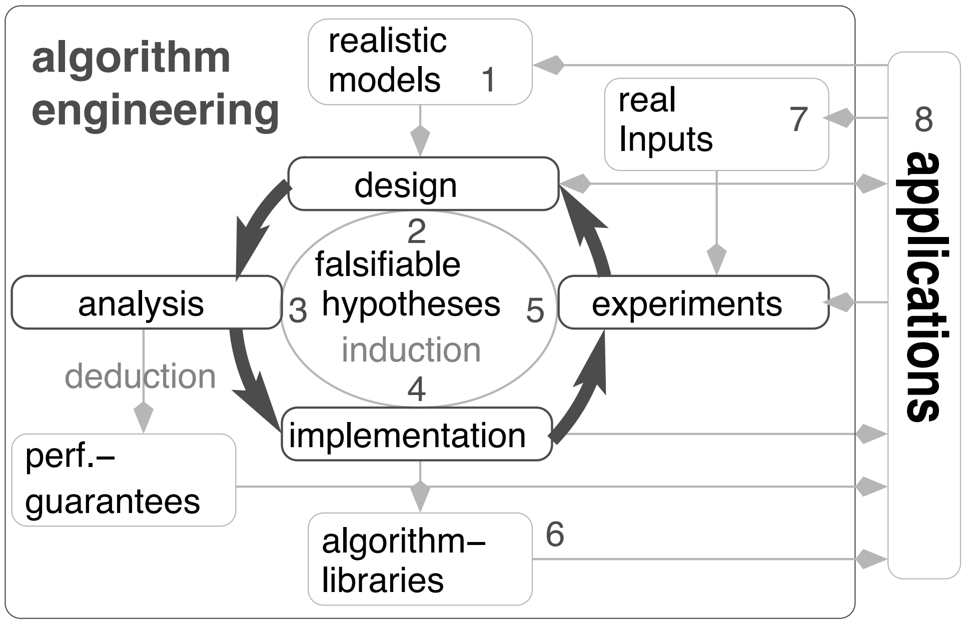


Figure 2.2: Representation of Algorithmics as Algorithm Engineering.

if the execution of the best sequential algorithm is $T(n)$, then the execution of the parallel algorithm with $k > 1$ *processors* must at least be as fast $T_k(n) \leq T(n)$. If this minimal requirement is not fulfilled the hypothesis will not hold, and the algorithm will be rejected. No *asymptotic performance* and *practical efficiency* will be measured.

2.3 Asymptotic performance measures

The studied algorithms will be described in pseudocode and represented visually as *directed acyclic graph (DAG)* as in [14, 8]. In order to understand which parts of the algorithms are parallelizable in a DAG the following terms must be defined:

DAG: A directed acyclic graph, $G = (V, E)$, in this thesis representing the computations of an algorithm. The vertices, V, represent the instructions to be computed and the edges, E, represent the dependencies between the instructions. If there is a edge, $(u, v) \in E$, the instructions in the vertex v cannot be executed before the instructions in the vertex u.

Work: The total time of computing all instructions of the algorithm executed on a computer with a single processor. Work can be mapped to the DAG as all the vertices, V, in the graph. The *work law* states that if k work can be done in $T_k(n)$ time on a computer with *k processors* and the amount of work is only $T(n)$, where $k \cdot T_k(n) \geq T(n)$, then:

$$k \cdot T_k(n) \geq T(n) \implies \tag{2.1}$$

$$\frac{k \cdot T_k(n)}{k} \geq \frac{T(n)}{k} \implies \tag{2.2}$$

$$T_k(n) \geq \frac{T(n)}{k} \tag{2.3}$$

The execution time of the algorithm with *k processors*, is limited to the amount of work divided by the amount of processors.

Span: The execution time of the longest path in a DAG, also know as *critical path*. The asymptotic time complexity of the algorithm is dominated by the *critical path*. The *span law* is defined as follows: no parallel computer with a limited amount of *k processors* can execute a parallel algorithm faster than a computer with an unlimited amount of cores. The computer with unlimited amount of processors can always just execute the parallel algorithm with *k processors*:

$$T_k(n) \geq T_\infty(n) \tag{2.4}$$

Algorithm 2.3.1: `Foo`(A, p, r)

```
Data: Let $A$ be an array.
Data: Let $p$ and $r$ be indexes in the array $A$ such that $p \leq r$.
if $p < r$ then
    $q \leftarrow \frac{r-p}{2}$;
    Foo($p, q$);
    Foo($q, r$);
```

An example of pseudocode and a visual representation of a parallel algorithm executed with one processor and *k processors* represented as *DAGs* can be seen in Listing 2.3.1 and Figure 2.3. By looking at the pseudocode, the only operation that the algorithm does is to split up in half recursively. The split operation can be done in *constant time*, $O(1)$. The amount of operations that will performed at each stage of the recursion tree is $O(1) + ... + O(\frac{n}{2}) + O(n)$ respectively where $O(n)$ dominates. So the *work* for the algorithm is $O(n)$.

The *span* in the first DAG, which is executed sequentially, is of length n because no vertices can be visited at the same time. In the second DAG, because all non overlapping operations can be executed in the same time unit at each level of the recursion tree, the *span* will only be of length $\lg n$. Executing the algorithm sequentially will be in $T(n) = O(n)$ and executing the algorithm in parallel with k *processors* will be in $T_k(n) = O(\lg n)$ when $k \geq n$. We have now set a lower bound on the running time for the algorithm on a parallel computer with k *processors*.

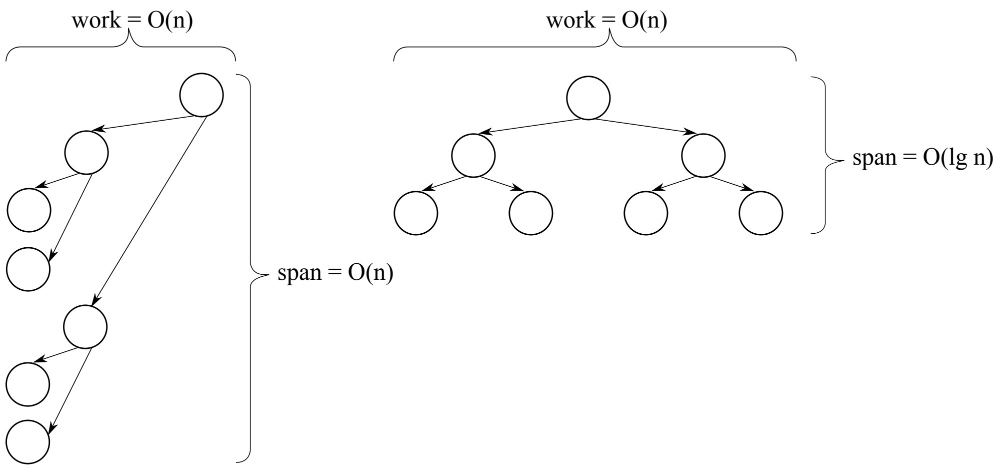


Figure 2.3: Representation of `Foo` sequential and parallel computation as *DAGs*.

2.4 Practical efficiency measures

The performance measures that will be used to show that a designated parallel algorithm is efficient are:

Speedup: A Comparison with the best sequential implementation.

$$speedup(k) = \frac{T(n)}{T_k(n)} \tag{2.5}$$

where k is the number of processors, and $T(n)$ is the time used to execute the best know sequential algorithm. This algorithm does not have to be the same as the parallel version, and $T_k(n)$ is the time used to execute the parallel algorithm with k processors.

Efficiency: Useful work performed on the available resources.

$$\textit{efficiency}(k) = \frac{P_i(k)}{P_m(k)} = \frac{\frac{T(n)}{k}}{T_k(n)} = \frac{T(n)}{k \cdot T_k(n)} = \frac{\textit{speedup}(k)}{k} \tag{2.6}$$

where k is the number of processors, $P_i(k)$ is the ideal performance and is defined as the best sequential time, $T(n)$, divided by the k processors and $P_m(k)$ is the actual performance and equals the time used to execute the parallel algorithm with k processors, $T_k(n)$.

Scalability: How increased resources provide a better performance.

$$\textit{scalability}(k) = \frac{T_1(n)}{T_k(n)} \tag{2.7}$$

where k is the number of processors, and $T_1(n)$ is the time used to execute the parallel algorithm with *1* processor and $T_k(n)$ is the time used to execute the parallel algorithm with k processors.

From these parameters a hypothesis will be presented for measuring performance on the given architectures. A special consideration for the calculation of *speedup* is to have in mind:

Amdahl's law [1, 12]: The maximum speedup of a parallel application is bounded to $\textit{speedup}(k) = \frac{1}{T_s(n)}$, where k is the amount of processors. Note that k does not affect the calculation of the maximum possible speedup. $T_s(n)$ is the time spent computing the sequential part of the program and $T_p(n)$ is the time spent computing the parallelizable part of the program. Assuming that both the sequential time and the parallelizable time executed on a sequential processor is:

$$T(n) = T_s(n) + T_p(n) = 1 \tag{2.8}$$

where the parallel time can be defined as:

$$T_p(n) = 1 - T_s(n) \tag{2.9}$$

$$T(n) = T_s(n) + 1 - T_s(n) \tag{2.10}$$

running the parallel part from (2.8) with k processors:

$$T_k(n) = T_s(n) + \frac{T_p(n)}{k} \tag{2.11}$$

$$= T_s(n) + \frac{1 - T_s(n)}{k} \tag{2.12}$$

looking at (2.5), the calculation of *speedup* for k processors with (2.8) and (2.12) would be:

$$speedup(k) = \frac{T(n)}{T_k(n)} = \frac{1}{T_s(n) + \frac{1-T_s(n)}{k}} \tag{2.13}$$

by setting k to infinity:

$$\lim_{k\to\infty} \left(\frac{1 - T_s(n)}{k} \right) = 0 \tag{2.14}$$

by combining (2.13) and (2.14):

$$speedup(k) = \frac{1}{T_s(n) + 0} = \frac{1}{T_s(n)} \tag{2.15}$$

An example, given an algorithm A where 30% of the algorithm runs in sequential time cannot be parallelized and 70% can be parallelized. Taking into consideration *Amdahl's law* a maximum speedup bound would be set to:

$$speedup(k) = \frac{1}{T_s(n)} = \frac{1}{\frac{30}{100}} = \frac{1}{\frac{3}{10}} = \frac{10}{3} \approx 3.33 \tag{2.16}$$

Amdahl's law is overruled when *superlinear speedup* is achieved, an example is when a problem can be split up so it fits in the *L1 cache* of each of the available processors bypassing the ideal performance $P_i(k) = \frac{T(n)}{k}$.

Framework overhead: On the other hand, some of the studied *C++ frameworks* have some initial, $T_{io}(n)$, and/or final, $T_{fo}(n)$, overhead that can affect the *total completion time* [19], $T_{tct(k)}(n)$, of the algorithm:

$$T_{tct(k)}(n) = T_k(n) = T_{io}(n) + T_s(n) + \frac{T_p(n)}{k} + T_{fo}(n) \tag{2.17}$$

performing (2.14) on (2.17)

$$speedup(k) = \frac{T(n)}{T_k(n)} = \frac{T(n)}{T_{tct(k)}(n)} \tag{2.18}$$

$$= \cdots = \frac{1}{T_{io}(n) + T_s(n) + T_{fo}(n)} \tag{2.19}$$

this will also affect on the maximum speedup bound.

Chapter 3

Existing frameworks and libraries

In this chapter several *C++ parallel frameworks and libraries* will be described with emphasis on how they allow parallelism, handling of shared data structures and the programming language syntax. As in [21], a few words describing the libraries and an example on of how to parallelize a sequential algorithm, in this case based on the recursion example from the previous chapter, will be presented for each library. The implementation in *C++* of the sequential version of the algorithm, in Listing 2.3.1, can be seen in Listing 3.1.

3.1 POSIX threads (pthreads)

The *Portable Operating System Interface for UNIX Threads* is a *C/C++ library* that offers parallelism through *threads*. As mentioned in Section 1.2,

```
template<typename RandomAccessIterator>
void foo(RandomAccessIterator first, RandomAccessIterator last){
  if(first == last){ return; }
  RandomAccessIterator middle(first+((last - first) >> 1));
  foo(first,middle);
  foo(middle,last);
}
```

Listing 3.1: Simple recursion in a sequential algorithm

a *process*, can create *threads* to execute instructions concurrently. The *API* offers two types of threads:

System thread: Whenever a *process* creates a *system thread*, the operating system will considered it as a peer to the creating *process* and the thread will compete for resources with all the other processes running on the system. System threads within a *process* or another *system thread* can execute concurrently on several processors, taking advantage of multiple processors architectures. *PTHREAD_SCOPE_SYSTEM* must be given as a parameter in order to create a thread as a system thread when calling *pthread_create*.

User thread: In difference to system threads, user threads within the same *process* or *system thread* can never be executed concurrently on other processors. This makes *user threads* unsuitable for parallelism. In order to create a user thread, *PTHREAD_SCOPE_PROCESS* must be given as a parameter when calling *pthread_create*.

The sequential code, in Listing 3.1, can be parallelized by doing a few modifications. The function and data types must be transformed to *(void *)* types because the function that creates a thread, *pthread_create*, only allows this type to be sent as a parameter. Data must be *casted* back to the class in order to use it. The programming syntax used to write the parallel version of the code is *pure C++*.

The problem in the code, in Listing 3.2, is that if there only is a limited number of threads available. Optimal usage of a system thread is usually *1-to-1* with a core, as mentioned in Section 1.2, but the recursive function will create one thread for each iteration reaching very fast the operating systems limit. The minimal value for *_POSIX_THREAD_THREADS_MAX* is 64 and it is defined in *basedefs/limits.h*. A way to avoid this is by using a thread pool. A thread pool will create a limited number of threads and instead of destroying the threads when tasks are done, the thread pool will reuse them to execute other tasks. This is mentioned in [23].

The mechanism presented for handling shared data structures are *mutex*, abbreviation for *mutual exclusion* and *condition* variables. By using a mutex when writing to data structure, no other thread will be able to read or write until the mutex is released. This gives *atomicity* when changing the state of a data structure and ensures no *race condition* will arise. A *condition* variable allows a thread to wait until a mutex for a shared data structure is released by the thread holding the lock. This will result in less computational

```
#include <pthread.h>
template<typename RandomAccessIterator>
class Range{
  RandomAccessIterator first;
  RandomAccessIterator last;
  Range(RandomAccessIterator first_, RandomAccessIterator last_)
    : first(first_), last(last_) {}
};

template<typename RandomAccessIterator>
void * foo(void * data){
  Data<RandomAccessIterator> *
    d(static_cast<Data<RandomAccessIterator> * >(data));
  if(d->first == d->last){ return; }
  pthread_t thread;
  RandomAccessIterator middle(d->first+((d->last - d->first) >> 1));
  pthread_create(&thread, 0, &foo<RandomAccessIterator>,
                 (void *) new Range<RandomAccessIterator>(d->first,d->
                    middle));
  foo(d->middle,d->last);
  pthread_join(thread, 0);
}
```

Listing 3.2: Parallel implementation of the sequential code with POSIX

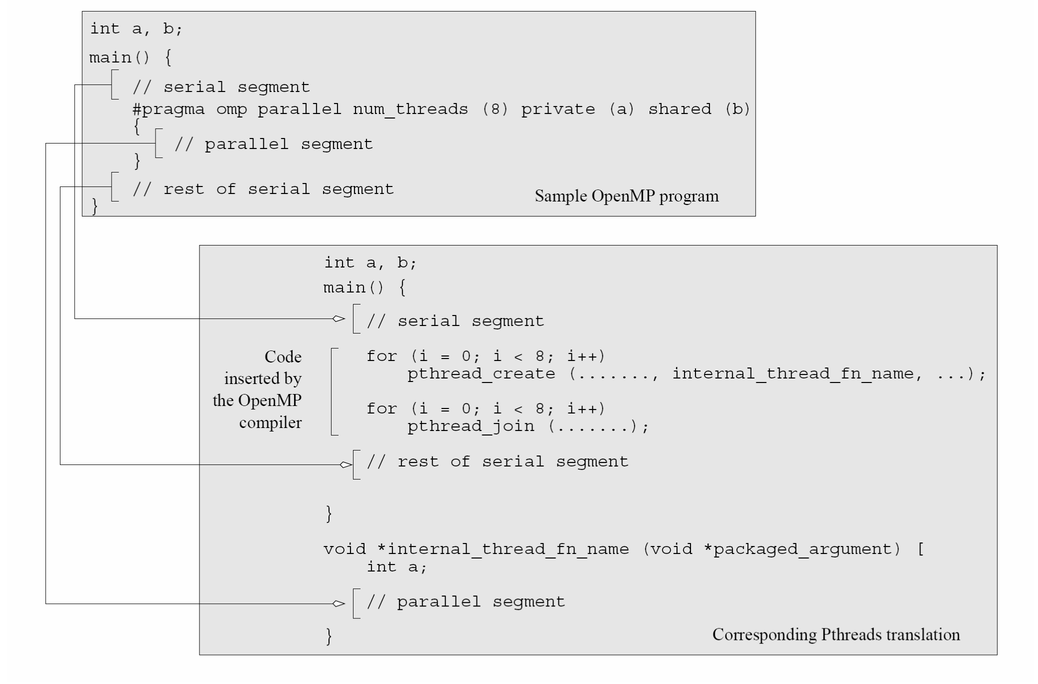
overhead. A side effect of using *mutex* is that *deadlocks* can arise when a thread t_1 is holding a lock on a data structure A and waiting for the data structure B while a thread t_2 is holding a lock on the data structure B and waiting for the data structure A. In this library there is no mechanism to detect *deadlocks*, it is the responsibility of the developer to avoid them.

Synchronization between threads can be achieved by using *barriers*, where a thread is not allowed to pass the barrier until the condition of the barrier is fulfilled. The main problem with *POSIX* barriers is that the implementation is optional, this means that a code relying on this form of synchronization might not be able to compile on different systems. Another way to implement synchronization is with the use of *pthread_join*. This function ensures that no code will be executed until the created thread is terminated with *pthread_exit*. It will only work if the thread is created in a *joinable* state.

3.2 OpenMP

The *Open Multi-Processing*, is a *C, C++ and Fortran API* for shared memory programming. OpenMP is implemented in all major compilers[1] and achieves

[1]GNU Compiler Collection (GCC), Low Level Virtual Machine (LLVM), Intel C++ Compiler (ICC) and Visual C++



```
int a, b;
main() {
    // serial segment
    #pragma omp parallel num_threads (8) private (a) shared (b)
    {
        // parallel segment
    }
    // rest of serial segment
}
```

```
int a, b;
main() {
    // serial segment

    for (i = 0; i < 8; i++)
        pthread_create (......., internal_thread_fn_name, ...);

    for (i = 0; i < 8; i++)
        pthread_join (.......);

    // rest of serial segment

}
void *internal_thread_fn_name (void *packaged_argument) [
    int a;

    // parallel segment
}
```

Code inserted by the OpenMP compiler
Corresponding Pthreads translation

Figure 3.1: Transformation from OpenMP compiler directives to C++ code.

parallelism through a set of *compiler directives (#pragma)*. An example of how this preprocessing compiler directives would look like if *pthreads* were used can be seen in Figure 3.1[2]. The parallelism strategy of the library built on a *hierarchical thread model* where a main thread creates a set of children treads, knows as *forks*. The main thread will wait until all the children threads are finished with their work, named *join*. The sequential code from Listing 3.1, can be parallelized with a few modifications. What stands out looking in the parallelized code, in Listing 3.3, is that parallelization in *OpenMP* is mostly based on *for loops*. One thing to have in mind when using this type of parallelization is that this will fork the threads in *linear time* $O(n)$. A way to achieve this in *logarithmic time* is to use nested *for loops* where the iterations are limited to *2*. The syntax of the code is *pure C++*.

The thread limit is once again an issue. By comparing the codes from Listing 3.2 and 3.3 and based on Figure 3.1. OpenMP will actually create twice the amount of threads than POSIX reaching earlier the operating system maximum limit of system threads. The problem could be solved in POSIX by using a thread pool. This solution is not possible for OpenMP.

[2]http://www.cs.aau.dk/∼adavid/teaching/MTP-06/11-MVP06-slides.pdf

```
template<typename RandomAccessIterator>
#include <omp.h>
void foo(RandomAccessIterator first, RandomAccessIterator last){
  if(first == last){ return; }
  RandomAccessIterator middle(first+((last - first) >> 1));
  #pragma omp parallel for
  for(typename iterator_traits<RandomAccessIterator>::value_type i(0);
      i < 2; ++i){
    if(i == 0){ foo(first,middle); }
    else{ foo(middle,last); }
  }
}
```

Listing 3.3: Parallel implementation of the sequential code with OpenMP

A set of compiler directives are given in order to work with shared data structures. By adding *shared(x)* to the compiler directive *#pragma for*, all the forked threads will then have access to the data structure *x*. An example can be seen below:

```
int bar [] = { 42 };
#pragma omp parallel for shared(x)
for(int i(0); i < n; ++i){
  foo(bar);
}
```

Synchronization is done when all forked threads are finished and the parent threads join them. Another way to achieve synchronization between threads is to use the compiler directive *barrier*. The barrier will ensure that until all forked threads reach that point, the execution of the parallel code will not be resumed.

3.3 Cilk++

The *Cilk++* library [18] is an extension to the *C++* programming language based on *MIT's* parallel programming language, *Cilk*, created among other by *Professor Charles E. Leiserson*, co-author of [8]. Even though the library is commercial software, property of *Intel*, the source code is available under an open source licence[3].

The library offers parallelism through three keywords: *cilk_for*, *cilk_spawn* and *cilk_sync*. This three primitives are mapped directly to *parallel*, *spawn* and *sync* in [8]. With these keywords, parallelizing sequential code is extremely easy. The code from Listing 3.1, is parallelized, in Listing 3.4, under less than a minute in a very intuitive way. The code is almost identical but

[3]http://sourceforge.net/projects/cilk/

```
template<typename RandomAccessIterator>
#include <cilk.h>
void foo(RandomAccessIterator first, RandomAccessIterator last){
  if(first == last){ return; }
  RandomAccessIterator middle(first+((last - first) >> 1));
  cilk_spawn foo(first,middle);
  foo(middle,last);
  cilk_sync;
}

int cilk_main(int argc, char *argv[]){
  int bar [] = { 42 };
  foo(bar+0,bar+1);
}
```

Listing 3.4: Parallel implementation of the sequential code with Cilk++

in order to compile it a specialized compiler must be used[4]. The syntax of the code is not *pure C++*.

In order to work with shared data structures, *Cilk++* introduces *hyperobjects*, which allows several threads to update the data structure maintaining the sequential order and ensuring no race conditions. The *hyperobjects* available are *reducers* which offers the possibility to append elements to a list or a string, basic logic operations and the possibility to find the maximum or minimum over a set of values. All other types of shared data structures are handled with *mutex* as in the *pthread* library. A visual application named *cilkscreen* can be used to detect *race conditions*.

As mentioned before, the created applications will only compile with the specialized compilers and only if the main function in the applications is renamed to *cilk_main*. This will restrict portability, if only a minor part of an application needs to be parallelized. Because of this, the whole application must be made parallelizable. Looking at the manual[5], merging *C++* and *Cilk++* libraries is not a straightforward task.

In [2, 18] the scheduler for multi-threaded computations used in Cilk++ is proved to execute an application that is suited for parallelization with $T(n)$ work and $T_\infty(n)$ span in an expected running time of:

$$T_k(n) \leq \frac{T(n)}{k} + O(T_\infty(n)) \qquad (3.1)$$

The utilized scheduler, *work stealing*, is based on a greedy scheduling technique where it is the inactive processors that try to steal work from the active

[4]Intel offers two compilers: One based GNU Compiler Collection (GCC) for Linux systems and the other based on Visual C++ for Microsoft systems.

[5]http://software.intel.com/en-us/articles/download-intel-cilk-sdk

processors. This differ from the conventional schedulers, *work sharing*, whenever a processor creates new threads, the scheduler tries to migrate them to other inactive processors.

By recalling the *work law* (2.3) and *span law* (2.4) the optimal execution time is:

$$T_k(n) \geq \frac{T(n)}{k} \implies T_k(n) = \frac{T(n)}{k} \tag{3.2}$$

The optimal span is:

$$T_k(n) \geq T_\infty(n) \implies T_k(n) = T_\infty(n) \tag{3.3}$$

And the ratio of *parallelism* between *work* and *span* is:

$$\frac{T(n)}{T_\infty(n)} \tag{3.4}$$

The result of using (3.2) and (3.3) as upper bounds for an optimal greedy scheduler to execute a computation with $T(n)$ work and $T_\infty(n)$ span on a computer with k *processors*, as in (3.1), is:

$$T_k(n) \leq \frac{T(n)}{k} + O(T_\infty(n)) \tag{3.5}$$

With (3.5) and if the ratio of *parallelism* (3.4) exceeds the number of k processors with a sufficient margin, at least 10 times more parallelism than processors, then a guarantee of a nearly *perfect linear speedup* is ensured. By rewriting the *work law* (2.3) as:

$$T_k(n) \geq \frac{T(n)}{k} \implies k \geq \frac{T(n)}{T_k(n)} \tag{3.6}$$

And combining (3.6) with (3.4). No *perfect linear speedup* can be achieved when the number of processors are greater than the ratio of *parallelism*

$$\frac{T(n)}{T_k(n)} \leq \frac{T(n)}{T_\infty(n)} < k \tag{3.7}$$

We assumed that the ratio of *parallelism* (3.4) exceeds the number of k processors with a sufficient margin

$$\frac{T(n)}{T_\infty(n)} \gg k \implies T_\infty(n) \ll \frac{T(n)}{k} \tag{3.8}$$

By combining (3.5) with (3.8) the term $\frac{T(n)}{k}$ dominates. The running time is:

$$T_k(n) \approx \frac{T(n)}{k} \tag{3.9}$$

And a nearly perfect linear speedup is achieved:

$$k \approx \frac{T(n)}{T_k(n)} \tag{3.10}$$

3.4 C++CSP2

The *C++ Communicating Sequential Processes 2* library is the only reviewed library that has a *network* model and not a *shared-memory* model like the other mentioned frameworks. The library is based on *CSP* [13] as the name indicates. The basics of *CSP* are that each process must have its own encapsulated data and algorithms and the only way to communicate between processes is by sending synchronized messages over channels. In order to share readable and writable data with this library, channels usage must be avoided. As channels work, it is only possible to send data structures as readable copies, *const T &*, which ensures that the data can not be modified by another process. A parallelization of the sequential code, in Listing 3.1, can be seen in Listing 3.5. The syntax of the code is *pure C++*. Two main problems in this way of implementation is that as in *pthreads* and *OpenMP* the limit of available system threads are reached very soon. Even though the library allows to create *user threads*, *RunInThisThread*, this will restrict the created threads only to run on a single processor. The other main problem is that, if channels are not used, then there is no longer any mechanism to avoid *race conditions*, which will make this type of implementation useless.

3.5 MCSTL

The *Multi-Core Standard Template Library* [15] is a parallel implementation of the *C++ Standard Template Library (STL)*. It makes use of multiple processors with shared memory and it is based on the previously described *OpenMP* API. Most of the algorithms from the *STL* are implemented. Data Structures *heaps* and the *STL containers* are not implemented. Since 2008 the library has been part of the *GNU Compiler Collection (GCC 4.3)*. The

```
#include <cppcsp/cppcsp.h>
template<typename RandomAccessIterator>
class Foo : public CSProcess{
private:
  RandomAccessIterator first;
  RandomAccessIterator last;
protected:
  void run(){
    if(first == last){ return; }
    RandomAccessIterator middle(first+((last - first) >> 1));
    Run( InParallel
         ( new Foo<RandomAccessIterator>(first,middle) )
         ( new Foo<RandomAccessIterator>(middle,last) ) );
  }
  public:
    Foo(RandomAccessIterator first_, RandomAccessIterator last_)
      : first(first_), last(last_) {}
  };
template<typename RandomAccessIterator>
void foo(RandomAccessIterator first, RandomAccessIterator last){
  Start_CPPCSP();
  Run( new Foo<RandomAccessIterator> (first,last) );
  End_CPPCSP();
}
```

Listing 3.5: Parallel implementation of the sequential code with C++CSP2

library is very straightforward. For example an application that uses the *STL sort* function can be refactored by changing only a few things, outcommenting *using std::sort* and adding *using std::__parallel::sort* and some extra parameters to ensure parallelism. An example of this can be seen in Listing 3.6.

As mentioned before, *MCSTL* is based on *OpenMP* and hereby inherits the problem of the operating system thread limit. In the implementation of their *quicksort* algorithms two solutions are presented. The first solution will call the best sequential sort algorithm whenever the limit is reached, in Listing 3.7. Second a *load-balanced* version of the algorithm is implemented on the *work-stealing* technique [2], described in Section 3.3, where if one threads finish all its tasks it looks for more work randomly on other threads. If there is work to do, the thread will steal *half* of the *victim threads* work.

```
#include <algorithm>
#include <parallel/algorithm>
#include <parallel/settings.h>
#include <omp.h>
#include <algorithm>
//using std::sort;
using std::__parallel::sort;
using __gnu_parallel::force_parallel;

typedef __gnu_parallel::_Settings settings;

int main(int argc, char *argv[]){
  const int k(2);
  const int n(1 << 20);
  int * a(new size_t[n]);
  srand(time(0));
  for(size_t i(n); i--;){
    a[i] = (rand() % (n >> 1) + 1);
  }

  omp_set_dynamic(false);
  omp_set_num_threads(k); //available processors

  settings s;
  s.algorithm_strategy = force_parallel;
  settings::set(s);

  sort(a,a+n);

  return EXIT_SUCCESS;
}
```

Listing 3.6: MCSTL Parallel sorting application

```
.../gcc44_c++/parallel/quicksort.h

template<typename RandomAccessIterator, typename Comparator>
  void
  parallel_sort_qs_conquer(RandomAccessIterator begin,
                           RandomAccessIterator end,
                           Comparator comp,
                           thread_index_t num_threads)
  {
    ...

    if (num_threads <= 1){
      __gnu_sequential::sort(begin, end, comp);
      return;
    }

    ...
  }
```

Listing 3.7: MCSTL parallel quicksort calls sequential sort when limit of threads are reached

Chapter 4

EAD++ framework

In this chapter the framework developed under this Master's thesis will be described briefly in the introduction. Afterwards an analysis reasoning why the framework was created followed by the design and implementation in *C++*. Finally a detailed description of the *API* is presented and the sequential code example from Listing 3.1 will be parallelized.

The *Efficient Algorithm and Data Structures framework for C++ (EAD++)* is a *multi-threaded* based on a *shared-memory* model that offers:

- a limit on the usage of threads with the help of a *thread pool*;
- a mechanism that ensures that jobs with higher priorities are executed first. This is achieved with help of a *job pool*;
- automatic load-balance of jobs based on a *helping technique*;
- *atomicity* when changing the state of a shared data structure;
- a *pure and simple C++* application programming interface (API);
- no dependencies to certain compilers;
- a synchronization mechanism based on *global thread* and *job* barriers;
- little memory overhead, a *constant factor*, defined on initialization.

The framework is around *400* lines of code, including the *open source* licence. The implementation can be seen in Appendix A.2.

The practical efficiency measures, *speedup*, *efficiency* and *scalability*, from the benchmark test matched up with *Cilk++* and *MCSTL*. In some cases *EAD++* even outperformed the other frameworks. For more information on the benchmark tests, look at Chapter 8.

4.1 Analysis

Why create a new framework instead of using some of the previously described in Chapter 3?

The main problem is that frameworks such as *pthread*, *OpenMP* and *CPPCSP2* have no built in mechanism to limit the creation of threads. As showed with the recursion example, the limit of the operating system is reached very fast. *MCSTL* which is based on *OpenMP* solves this problem by calling a sequential version of the algorithm. This approach is not acceptable. If there are no available threads at the time when the sequential algorithm is called there might be some later on but because the algorithm now is sequential it cannot make use of the available threads. A framework should have a mechanism that can limit the usage of threads.

The way that *pthread* handles the sharing of data structures is by casting data of different types to *void pointers* and back to the given type. This approach removes the responsibility from the compiler, *C++* is a *strongly typed* language, and gives it to the developer. The errors will no longer be found at compiling time but at run time. This is not acceptable.

The synchronization mechanism must not be built on an optional specification of a library, as *barriers* in *pthread*. A clear definition must be made so executed code have the same behavior independent on the platform on which it is executed.

The presented *application programming interface (API)* must be simple, intuitive and *pure C++*. Writing code with *compiler directives* and performing all parallelism with *for loops* it is not very easy or intuitive. Even though *Cilk++* achieves parallelism with the help of only the following primitives: *cilk_for*, *cilk_spawn*, *cilk_sync* and *cilk_main*. The problem is that these primitives are not considered *pure C++* and are specific to *Intel's* version of the *GNU Compiler Collection (GCC)*.

Finally the presented framework must not be compiler specific as *MCSTL* and *Cilk++* are to the *GNU Compiler Collection (GCC)*.

4.2 Design and implementation

Based on the previous analysis, a *pure C++* framework needed to be created. The backbone of the framework is built on the *pthreads* library because it:

- allows to create *system threads* and hereby take advantage of a multi-core architecture;
- is implemented on most platforms such as *Unix*, *Linux* and even *Microsoft Windows*;
- has a mechanism to handle shared data structures;

The limitations of the library, as previously mentioned, are solved by introducing:

- a thread pool that limits the creation of threads;
- an abstract class, *Job*, that must be inherited in order to add work to the thread pool. Because there is only one *type* of work accepted by the thread pool, the responsibility of casting void pointers back to data is removed from the developers;
- *global thread* and *job* barriers in order to allow synchronization between threads;

Thread pool

The amount of threads in the thread pool are set on initialization. It is not possible to add or remove threads once the thread pool is running. All the threads created by the thread pool are *system threads*. Threads in the thread pool have three states: *active* when executing a job, *sleeping* when no job is available and *waiting* at a synchronization barrier. A thread sleeps by waiting on the job pool *condition variable*. A thread will be awaken when another thread sends a signal, *condition broadcast*, after adding a job to the job pool.

The way the thread pool *load-balances* the work between the threads is based on a *helping technique*. The *helping technique* is introduced in this thesis and it is based on that a thread will only create a job if there is at least one inactive thread. The created job is placed in the job pool and the inactive threads are notified. It is now the responsibility of the inactive threads to remove the job from the job pool and execute it. By using this technique, work can be distributed to *k inactive processors* in *logarithmic time*. Once all the processors are active, no more jobs will be created. This will reduce the amount of created jobs based on work. On other techniques such as *work stealing*, previously described, even though there are no inactive threads, the active threads will still create jobs because when other threads are finished with their tasks and they become inactive they need to be able to steal work from other threads. Assuming that all threads finish at the same time, all the resources and time used to create the stealable jobs are wasted.

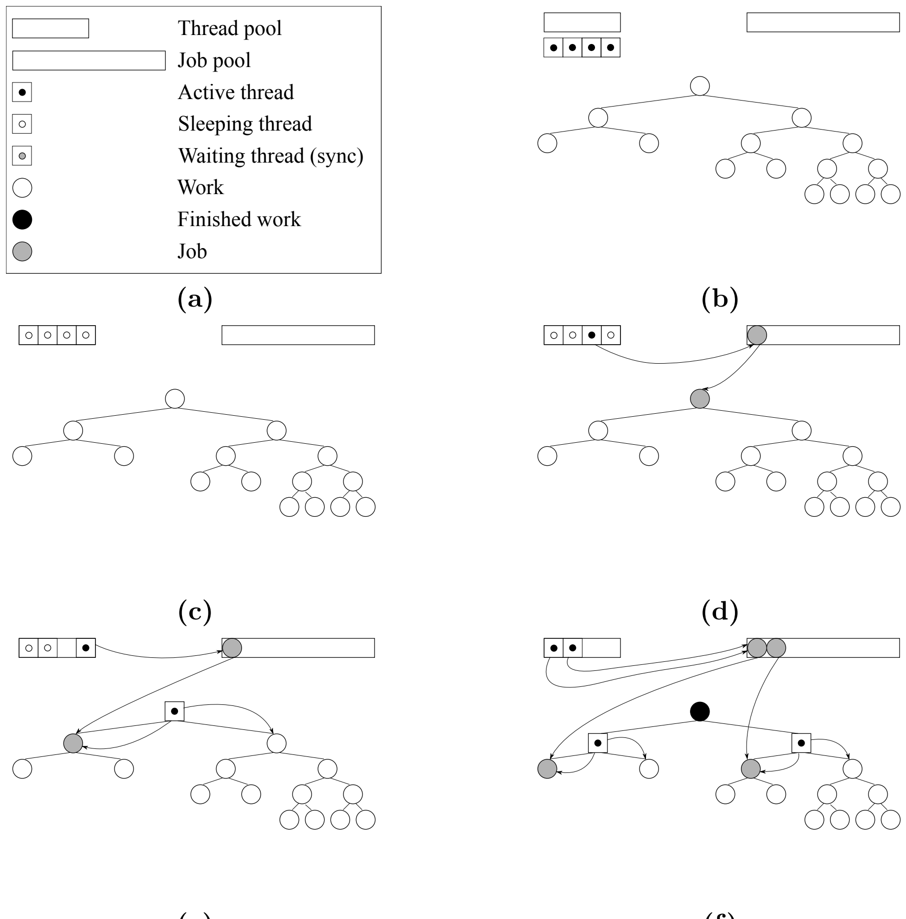


Figure 4.1: EAD++ helping technique.

A visual representation on how the thread pool works, can be seen in Figures 4.1 and 4.2. The thread pool is started with four threads and a job pool of size eight (Figure 4.1.b). All threads are active, they notice that the job pool is empty and go to *sleep* (Figure 4.1.c). The main thread adds a job and notifies all the threads from the thread pool. One of them will take the job and execute it (Figure 4.1.d). The thread will now check if other threads in the thread pool are *sleeping*. In this case, three threads are *sleeping*. The thread adds a job from the left branch of the graph and continues to work on the right side of the graph (Figure 4.1.e). This will be repeated until all threads are active (Figure 4.1.f). After all threads are active there are no need to create more jobs (Figure 4.2.b). Once again, when there are *sleeping* threads, new jobs will be added to the job pool (Figure 4.2.c and 4.2.e).

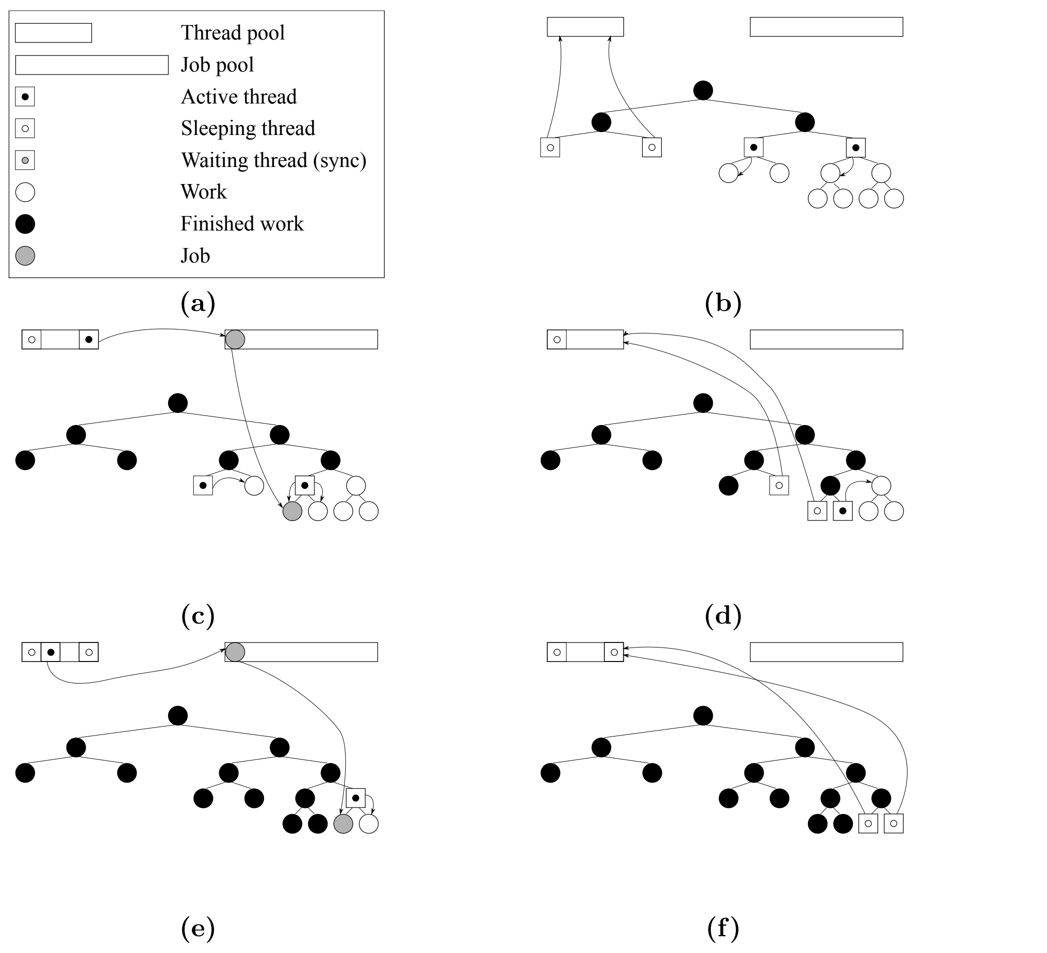


Figure 4.2: EAD++ helping technique.

Job pool

The job pool is implemented as a *stack (LIFO)*, where the last job arrived is the first one served. The underlying container is a *vector from the C++ STL*. Because of *polymorphism*, the job pool must be implemented as vector of job pointers. The memory used for the job pool is reserved when the thread pool is instantiated and because no dynamic memory allocation is used, the jobs can be added and removed in *constant time* $O(1)$. The job pool can also be used as a *priority queue*. This can be achieved by setting the parameter *heap* to true when instantiating the thread pool. Jobs, based on their priority, can be added and removed from the *heap* in *logarithmic time*. Because the underlying data structure is a *min-heap*, jobs with lower priorities will be served first. The behavior of a *FIFO* can also be achieved if created jobs have a higher priority then the job creating them. If not specified the initial size of the job pool is set to the amount of available threads in the thread pool. No dynamic memory allocation is allowed and it is defined on initialization, *constant factor*.

Shared data structures

The change of state of a shared data structure is handled with help of *mutex* and *condition* as in *pthreads*. This will ensure *atomicity*. Note that *Cilk++* only allows the usage of *mutex*. It is the developers responsibility not to create *deadlocks*, and as in the other mentioned frameworks, there is no mechanism to detect them.

Synchronization

Synchronization can be done in the form of two barriers. The first one is a *global thread barrier* that should only be called from the *main thread* of the application. This barrier ensures that all threads that are part of the thread pool will have finished their work and that the job pool is empty before resuming the execution of the code that follows the barrier. If the global barrier is called from a thread belonging to the thread pool, the condition will never be fulfilled because there will always be an active thread waiting at the barrier. Correct usage of the barrier from the main thread is the following:

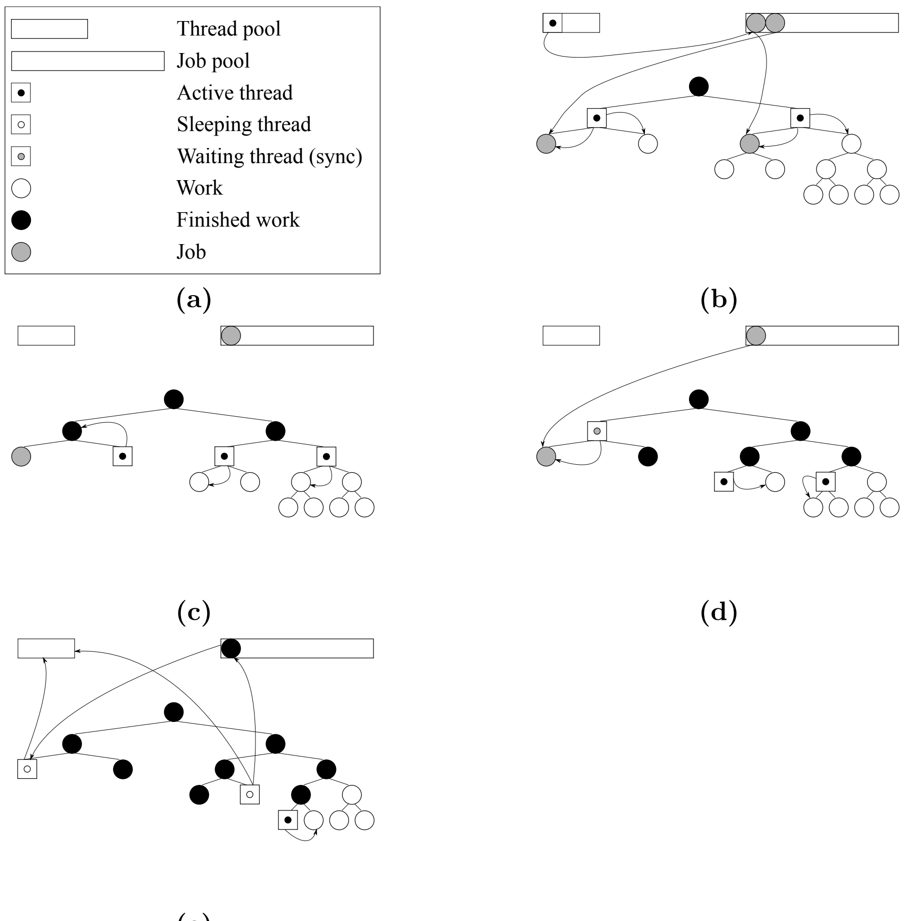


Figure 4.3: EAD++ job barrier.

```
{
  ...
  gtp_init(nr_threads, heap, buffer);
  shared_ptr<Job> j1(new Foo<RandomAccessIterator>(first, middle));
  gtp_add(j1);
  gtp_sync_all();
  shared_ptr<Job> j2(new Foo<RandomAccessIterator>(middle, last));
  gtp_add(j2);
  gtp_sync_all();
  gtp_stop();
  ...
}
```

The other type of barrier is the *job barrier*. This barrier ensures that a thread will wait at that point until the job given as a parameter is finished. If no other thread is executing the job. The waiting thread will execute the job. This way the waiting time at *job barriers* is minimal.

A visual representation can be seen in Figure 4.3. The active threads add a job to the job pool because they notice that a thread is sleeping. The awoken thread retrieves the a job from the job pool. There are now three threads active and one job in the job pool (Figure 4.3.c). The thread performing task in the left branch of the graph reached a leaf and goes one level up waiting for the other leaf to be done (Figure 4.3.d). The two other threads are busy performing a task on the right side so they are not able to help. The left thread, instead of waiting for the other threads to do the job, does it itself. Once the job is done the threads go back to sleep, but it does not remove the job from the job queue, that task will be performed by another thread whenever it checks if there are more jobs available (Figure 4.3.e).

The waiting thread can access the unfinished job in the job pool in *constant time* because the barrier has a pointer to the job. The implemented pointers are not standard pointers but *std::tr1::shared_ptr*[1]. This type of pointer is used because the job pointed at is guaranteed to be deleted when the last *shared_ptr* pointing to the job is destroyed. The only compiler that does not have implemented *TR1*'s shared pointer is *Intel's C++ Compiler (ICC).* This can be solved by adding the implementation of the *boost* library instead.

4.3 API

The *application programming interface* presents two interfaces: a *global thread pool* and a *local thread pool*. The *global thread pool* is presented in order to reduce the overhead of parallelizing an algorithm by allowing functions to call a global variable instead of having to add an extra parameter, the thread pool, to all the functions.

Global thread pool

gtp_init(size_t threads, bool heap, size_t buffer): The function shuts down an already global thread pool, if instantiated, and creates a new thread pool with the given parameters. If heap is set to true the job pool will be a *min-heap*. If the buffer value is given, the size of the job pool will be the amount of threads plus the buffer value.

gtp_add(shared_ptr<Job> job): Adds a job to the job pool. The job will be executed by one of the threads from the thread pool.

[1]Smart Pointers (TR1 document nr. n1450): http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1450.html

gtp_add_optimized(Function f, Parameters ps): A job is only created if there is at least one sleeping thread. Otherwise the work will be executed by the thread calling the function. This function is only supported by compilers that have implemented the *experimental* version of the upcoming *C++0x*.

gtp_add_optimized_args_*number*(F f, P1 p1,... , PN pN): The functionality is the same as the previous mentioned function. There are only implemented up to *three* arguments. If more arguments are needed, the framework should be expanded following the guidelines of these functions.

gtp_sync_all(): This barrier ensures that no code after the barrier will be executed until all threads in the thread pool have finished their work and no jobs are left in the job pool. **Remark:** *This barrier must **only** be called by the main thread that has instantiated the thread pool.*

gtp_sync(shared_ptr<Job> job): This barrier ensures that no code, after the barrier, will be executed until the job, given as parameter, is executed and finished.

gtp_stop(): The global thread pool is shutdown correctly.

Job class

Job(size_t priority): An abstract class that must be inherited in order to add jobs to the thread pool.

virtual run(): This function must be implemented. It is the function that the threads in the thread pool will execute.

priority(): Get the priority of a job.

set_priority(size_t n): Set the priority of a job.

status(): Get the status of a job.

done(): Set the status of a job to done when a job is finished.

Mutex and Condition

These are *C++ classes* wrapping the corresponding *pthread* procedure calls. These classes should be used when working with shared data structures to ensure atomicity.

```
template<typename RandomAccessIterator>
class Foo : public Job{
  RandomAccessIterator f,l;
  Foo(RandomAccessIterator first, RandomAccessIterator last)
    : Job(), f(first), l(last) { }
  void foo(RandomAccessIterator first, RandomAccessIterator last){
    if(first == last){ return; }
    RandomAccessIterator middle((last - first) >> 1);
    gtp_add_optimized<Foo<RandomAccessIterator> >(first,middle);
    foo(middle,last);
  }
  void run(){
    foo(f,l);
    done();
  }
};

template<typename RandomAccessIterator>
void bar(RandomAccessIterator first, RandomAccessIterator last,
         const size_t & nr_threads = 1, const bool & heap = false,
         const size_t & buffer = 0){
  gtp_init(nr_threads, heap, buffer);
  shared_ptr<Job> j(new Foo<RandomAccessIterator>(first, last));
  gtp_add(j);
  gtp_sync_all();
  gtp_stop();
}
```

Listing 4.1: Parallel implementation of the sequential code with EAD++

Local thread pool

The *local thread pool* shares the same interface as the *global thread pool* but without the prefix *gtp_*. The functions *gtp_init* are replaced by standard *C++* syntax when instantiating a class, *ThreadPool ltp(42, false, 0);*. The *gtp_stop* function is not necessary. The local thread pool will shut down correctly when the destructor is called. This occurs whenever the instance of the class goes out of scope.

An example on parallelizing the code from Listing 3.1, can be seen in Listing 4.1. Basically the only thing that has to be done is to encapsulate a given function into a class that inherits from the abstract class *Job*. A virtual function, *run*, must be implemented. A new function wrapper that instantiates the *thread pool*, in this case the *global thread pool* adding the first job. The rest of the jobs will be added recursively by the treads in the thread pool. Thanks to the *global thread* barrier no synchronization barrier needs to be added to the function *foo* as it was necessary for the framework *Cilk++* with *cilk_sync* and the *pthread* library with *pthread_join*.

CHAPTER 5

Sorting in parallel

In this chapter the following sorting algorithms: `Bitonicsort`, `Mergesort` and `Quicksort` will be studied with emphasis on how they can be parallelized. The sorting algorithms were chosen primary based on two criteria: the *worst-case* asymptotic time and minimal extra space utilization, in this order. A minimal requirement of being at least as fast as the best sequential algorithm, *C++ STL sort*, implemented by David R. Musser[1], a visualization of the sorting algorithm can be seen in 5.1. If the initial experiments show that the algorithm does not fulfill the minimal requirement, then the algorithm will be rejected and no *practical efficiency measures* will be made. The three chosen algorithms have in common that they can be implemented on a *Exclusive-Read Exclusive-Write (EREW) parallel random access machine (PRAM)*.

A requirement for the implementation in *C++* of the sorting algorithms is that they comply with the *C++ STL container* policy [25]. A function, in this case a sorting algorithm, will only take two *iterators* as input. The first *iterator* will point to the first element in the *container* and the second *iterator* will point to the last element in the *container*. This will allow the function to take different containers as input, disregarding of their *type*. This can be achieved with the use of *templates* [26]. No implementation details will be added to the pseudocode.

[1]http://www.cs.rpi.edu/~musser/

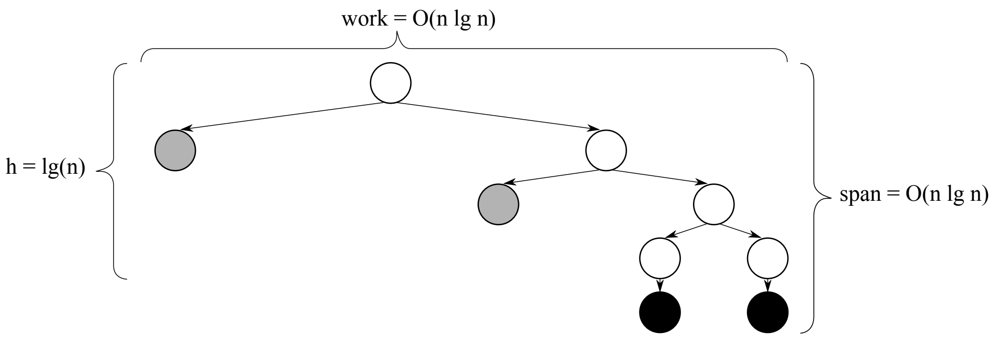


Figure 5.1: *C++ STL sort* also known as *introsort* is a variation of quicksort were the pivot to the partition is chosen with the *median-of-three* technique (white vertices), the large recursive calls are eliminated with help of *heap sort* (black vertices) and small subsequences, of at most size *16* are left unsorted (grey vertices). When the recursion returns, insertion sort is performed on the whole sequence in *chunks* of *16* elements. Worst-Case time complexity is in *log-linear time* $O(n \lg n)$.

5.1 Bitonicsort

`Bitonicsort` is a parallel sorting network algorithm, that sorts all input sequences in a constant number of comparisons which makes it data independent. The *work* is $\Theta(n \lg^2 n)$ and the *span* is $O(\lg n \cdot \lg n) = O(\lg^2 n)$. The *critical path* is calculated on the basis that it takes $\lg(n)$ stages, where $n = 2^m,\ m \in \mathbb{N}$, to to sort a sequence of n elements with *k processors*:

$$T_k(n) = \sum_{i=1}^{m} i = \frac{m^2 + m}{2} = \frac{\lg^2 n + \lg n}{2} = O(\lg^2 n) \tag{5.1}$$

Sorting the sequence can be done without using any extra space. A visual representation in form of a *directed acyclic graph* can be seen in Figure 5.2.

The algorithms is built on a set of network components:

`Comparator`. The network component, which has two input gates and two output gates, ensures that the minimal value of the two input values is set to the first output gate and the maximum value is set to the second gate. Pseudocode for the component can be seen in Algorithm 5.1.1.

`Half-cleaner`. The network component, which has $2^m,\ m \in \mathbb{N}$ input and output gates respectively, takes a *bitonic sequence* of numbers as

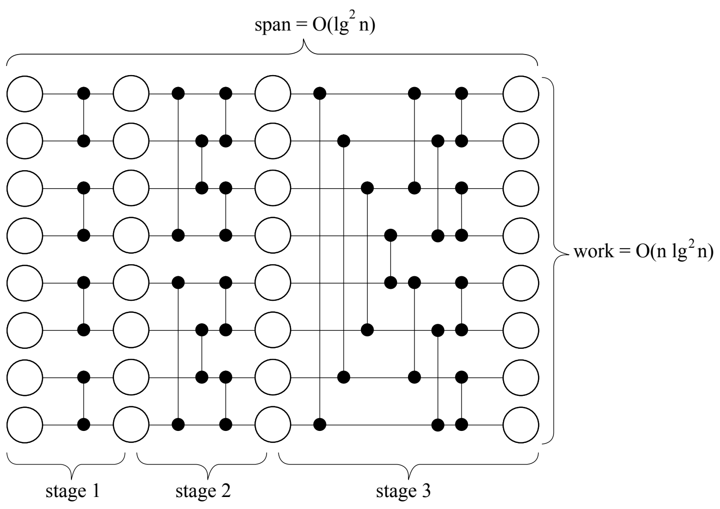


Figure 5.2: Representation of `Bitonicsort` computation as a DAG. Directed edges go from left to right.

Algorithm 5.1.1: `Comparator`(x, y)

```
Data: Let $x$ and $y$ be two arbitrary numbers.
1 $x' \leftarrow$ min$(x, y)$;
2 $y' \leftarrow$ max$(x, y)$;
3 return $(x', y')$;
```

input. A bitonic sequence is defined as a sequence that monotonically increases and then monotonically decreases, $\langle s_0 \leq \cdots \leq s_i \geq \cdots \geq s_n \rangle$ where $0 \leq i \leq n$, or monotonically decreases and then monotonically increases, $\langle s_0 \geq \cdots \geq s_i \leq \cdots \leq s_n \rangle$ where $0 \leq i \leq n$. The component ensures that all the elements in the top half of the outputted elements are less or equal to all the elements in the half bottom of the outputted elements. Both outputted halves are still bitonic sequences and the bottom is clean bitonic, every element is greater or equal to the first half of the output. Pseudocode for the component can be seen in Algorithm 5.1.2.

`Bitonic-sorter`. The network component, which has 2^m, $m \in \mathbb{N}$ input and output gates respectively, takes a *bitonic sequence* of numbers as input and produces a sorted sequence of numbers. The `Half-cleaner` network component is part of this component which ensures that the received bitonic sequence will be transformed into two bitonic sequences and from these two sequences the component calls itself recursively. Pseudocode for the component can be seen in Algorithm 5.1.3.

Algorithm 5.1.2: `Half-cleaner(`B`)`

```
Data: Let $B$ be a bitonic array.
Data: Let $n$ be the length of $B$. We assume that $n = 2^k$ where $k \geq 1$.
Data: Let $m$ be half the length of $B$, $\left(m = \frac{n}{2}\right)$
parallel
    for $i \leftarrow 1$ to $\frac{n}{2}$ do
        $(B[i], B[m]) \leftarrow$ Comparator($B[i], B[m]$);
        $i \leftarrow i + 1$;
        $m \leftarrow m + 1$;
sync;
```

Algorithm 5.1.3: `Bitonic-sorter(`B`)`

```
Data: Let $B$ be a bitonic array.
Data: Let $n$ be the length of $B$. We assume that $n = 2^k$ where $k \geq 1$.
Data: Let $m$ be half the length of $B$, $\left(m = \frac{n}{2}\right)$
Data: Let $B_1 = B[1, \ldots, m]$ be the first half of the bitonic array $B$ and let
      $B_2 = B[m + 1, \ldots, n]$ be the second half.
Half-cleaner($B$);
if $2 < n$ then
    parallel
        Bitonic-sorter($B_1$);
        Bitonic-sorter($B_2$);
    sync;
```

`Merger`. The network component, which has 2^m, $m \in \mathbb{N}$ input and output gates respectively, takes two sorted sequences of numbers as input and produces a sorted sequence as output. `Merger` is a recursive component that in its first stage produces to bitonic sequences. These sequences are sorted by calling the `Bitonic-sorter` afterwards. Pseudocode for the component can be seen in Algorithm 5.1.4.

`Sorter`. The network component, which has 2^m, $m \in \mathbb{N}$ input and output gates respectively, takes an arbitrary sequence of numbers as input and produces a sorted sequence of numbers. The component is recursive where in its first stage produces two sorted sequences that are given as input to the `Merger` that will produce one sorted sequence with all the numbers. Pseudocode for the component can be seen in Algorithm 5.1.4.

Algorithm 5.1.4: `Merger(`S_1, S_2`)`

```
Data: Let $S_1$ and $S_2$ be sorted arrays.
Data: Let $m_1$ be the length of $S_1$ and let $m_2$ be the length of $S_2$.
Data: Let $n$ be the length of $S_1$ and $S_2$, $(n = m_1 + m_2)$. We assume that
      $m_1$ and $m_2$ have the same length, $(m = m_1 = m_2)$ and that $n = 2^k$
      where $k \geq 1$.
1  parallel
2      for $i \leftarrow 1$ to $m$ do
3          $(S_1[i], S_2[n]) \leftarrow$ Comparator($S_1[i], S_2[n]$);
4          $i \leftarrow i + 1$;
5          $n \leftarrow n - 1$;
6  sync;
7  if $2 < n$ then
8      parallel
9          Bitonic-sorter($S_1$);
10         Bitonic-sorter($S_2$);
11     sync;
```

Algorithm 5.1.5: `Sorter(`A`)`

```
Data: Let $A$ be an array.
Data: Let $n$ be the length of $A$. We assume that $n = 2^k$ where $k \geq 1$.
Data: Let $m$ be half the length of $A$, $\left(m = \frac{n}{2}\right)$
Data: Let $A_1 = A[1, \ldots, m]$ be the first half of the array $A$ and let
      $A_2 = A[m + 1, \ldots, n]$ be the second half.
1  if $2 < n$ then
2      parallel
3          Sorter($A_1$);
4          Sorter($A_2$);
5      sync;
6  Merger($A$);
```

The only limitation to the algorithm is that it can only sort sequences of numbers with size 2^m, $m \in \mathbb{N}$. This size limitation is solved by adding three more components to the network. This does not change the worst-case asymptotic time complexity but it adds at most n in extra space usage:

`Find-max`. Given a sequence of size n, finding the maximum number can be achieved in $O(\lg n)$ time assuming that all non-overlapping operations are done in parallel. Pseudocode for the component can be seen in Algorithm 5.1.6.

Algorithm 5.1.6: `Find-max(`A`)`

```
Data: Let $A$ be an array.
Data: Let $n$ be the length of $A$.
Data: Let $A_1 = A[1, \dots, m]$ be the first half of the array $A$ and let
      $A_2 = A[m+1, \dots, n]$ be the second half.
1 if $1 \leq n$ then
2     return $A[1]$;
3 parallel
4     $x \leftarrow$ Find-max($A_1$);
5     $y \leftarrow$ Find-max($A_2$);
6 sync;
7 return max($x, y$);
```

`Populate`. By taking two sequences as input, A and B, it populates all the possible elements of A in B when $A \geq B$, otherwise it fills the remaining empty places with the number x, which is given as a parameter. The numbers can be populated in $O(\lg n)$ assuming that all overlapping operations are done in parallel. Pseudocode for the component can be seen in Algorithm 5.1.7.

`Sorter'`. Given a sequence A of size n where $n \neq 2^m,\ m \in \mathbb{N}$. Can be sorted with a bitonic sorter network by moving all the numbers into a new sequence B of size n' where $n' = 2^{\lceil \lg n \rceil}$ and populating the empty spaces in B with the maximum value from A. With the previous two components, this can be done in $O(\lg n)$ time. The sequence B can now inserted into the *bitonic sorting network*. Once the sequence B is sorted all the sorted values are moved back to the initial sequence A, in $O(\lg n)$ time by using the `Populate` component. Because none of the used components execution time are greater than $O(\lg^2 n)$, therefore it will still dominate the asymptotic time complexity of the algorithm and `Sorter'` will still be in $O(\lg^2 n)$. Pseudocode for the component can be seen in Algorithm 5.1.8.

Algorithm 5.1.7: `Populate`(A, B, b, e, x)

```
Data: Let $A$ and $B$ be two arrays.
Data: Let $n$ be the length of $A$. We assume that $n = 2^k$ where $k \geq 1$.
Data: Let $x$ be an arbitrary number.
Data: Let $b$ and $e$ be indexes in the array $B$ such that $b \leq e$.
Data: Let $d$ be the difference between the beginning and ending indexes,
      $(d = e - b)$. We assume that $d = 2^k$ where $k \geq 1$.
Data: Let $m$ be half the length of $d$, $\left(m = \frac{d}{2}\right)$
1  if $1 = d$ then
2      $B[b] \leftarrow x$;
3      return;
4      if $b < n$ then
5          $B[b] \leftarrow A[b]$;
6          return;
7  parallel
8      Populate($A, B, b, b + m, x$);
9      Populate($A, B, b + m, e, x$);
10 sync;
```

Algorithm 5.1.8: `Sorter'`(A)

```
Data: Let $A$ be an array.
Data: Let $n$ be the length of $A$.
Data: Let $k = \lceil \lg n \rceil$.
Data: Let $A'$ be a new array of length $n' = 2^k$.
1  $x \leftarrow$ Find-max($A$);
2  Populate($A, A', 0, n', x$);
3  Sorter($A'$);
4  Populate($A', A, 0, n, x$);
```

Implementation details

Implementing the algorithm as a sequential version was a straightforward task, except when transforming the parallelizable *for loops* into recursive calls. Implementing a parallel version based on the sequential code for both *Cilk++* and *EAD++* was an easy task. The sequential and parallel code can be seen in Appendix A.1.

The initial experiments showed that the parallel algorithm did not fulfill the minimal requirement no matter the size of the input or the amount of processors. The algorithm was rejected and there will not be performed *practical efficiency measures* with real data.

Even though in this study the implemented parallel algorithm did not perform as desired, in [17, 4] they show that by implementing the algorithm with *CUDA* and executing it on a *NVIDIA GeForce GTX 280*[2] with *240 multiprocessor cores*, the performance exceeds the best sequential algorithm. A conclusion can be made that until the number of processors available is in the same range as *NVIDIA GPUs*, this algorithm will not be useful.

5.2 Mergesort

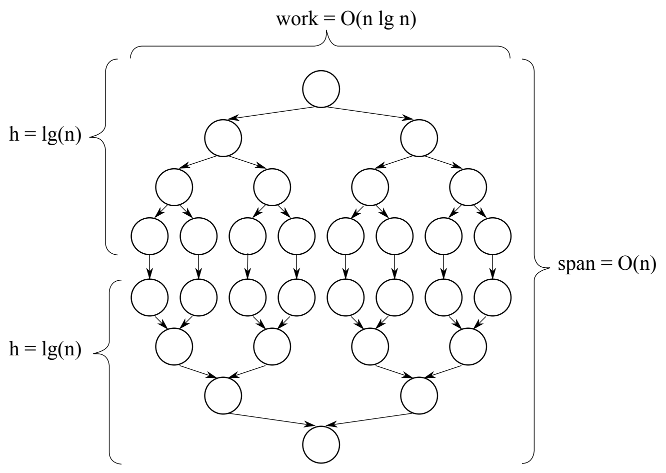


Figure 5.3: Representation of `Mergesort` computation as a DAG.

`Mergesort`, in Algorithm 5.2.3, is a comparison sorting algorithm, that sorts input sequences of n elements with $O(n \lg n)$ comparisons. The algorithm is based on the *divide-and-conquer* paradigm where a main problem is divided into subproblems of the same type until they are simple enough to be solved. In this particular algorithm the input sequence is split in half and the algorithm is called recursively on each half. This step is done until the subproblems are of size one. The two subsequences are merged together, in Algorithm 5.2.1, forming one single sorted sequence. The time complexity of the algorithm is in $\Theta(n \lg n)$ with n extra space. The extra space used is to allocate the temporary arrays in the `Merge` process, in Algorithm 5.2.1. Alternatives to minimize the use of extra space would be to compute the

[2]http://www.nvidia.com/object/product_geforce_gtx_280_us.html

merge process *in place*. A naive implementation is presented in Algorithm 5.2.2. The problem with this algorithm is that the time complexity is in $O(n^2)$. This term would dominate changing the time complexity of `Mergesort`. This is not acceptable. In [16] an *in-place mergesort* algorithm is presented that runs in $O(n \lg n)$.

Algorithm 5.2.1: `Merge`(A, p, q, r)

```
Data: Let $A$ be an array.
Data: Let $p$, $q$ and $r$ be indexes in the array $A$ such that $p \leq q \leq r$.
Data: Let $A_1$ and $A_2$ be two arrays of size $n_1 = q - p$ and $n_2 = r - q - 1$.
for $i \leftarrow 0$ to $n_1$ do
    $A_1[i] \leftarrow A[p + i]$;
for $i \leftarrow 0$ to $n_2$ do
    $A_2[i] \leftarrow A[q + 1 + i]$;
$i \leftarrow 0,\ j \leftarrow 0,\ k \leftarrow p$;
while $k < r$ do
    if $A_1[i] \leq A_2[j]$ then
        $A[k] \leftarrow A_1[i]$;
        $i \leftarrow i + 1$;
    else
        $A[k] \leftarrow A_2[j]$;
        $j \leftarrow j + 1$;
    $k \leftarrow k + 1$;
```

Algorithm 5.2.2: `Inplacemerge`(A, p, q, r)

```
Data: Let $A$ be an array.
Data: Let $p$, $q$ and $r$ be indexes in the array $A$ such that $p \leq q \leq r$.
while $p < q$ do
    if $A[p] > A[q]$ then
        $A[p] \leftrightarrow A[q]$;
        $q' \leftarrow q$;
        while $q' < r$ and $A[q'] > A[q' + 1]$ do
            $A[q'] \leftrightarrow A[q' + 1]$;
            $q' \leftarrow q' + 1$;
    $p \leftarrow p + 1$;
```

The algorithm can be parallelized because of its nature, *divide-and-conquer*.

Algorithm 5.2.3: `Mergesort`(A, p, r)

```
Data: Let $A$ be an array.
Data: Let $p$ and $r$ be indexes in the array $A$ such that $p \leq r$.
1 if $p < r$ then
2     $q \leftarrow \frac{\lfloor p+r \rfloor}{2}$;
3     Mergesort($A, p, q$);
4     Mergesort($A, q + 1, r$);
5     Merge($A, p, q, r$);
```

Every subproblem can be executed on a separate process. Using a barrier to ensure that both the subproblems are done before the two sequences are merged. The *work* is $\Theta(n \lg n)$ and the *span* is $O(n)$. The *critical path* is bounded to last merge because the cost of the partitions are in constant time:

$$T_k(n) = \sum_{i=1}^{\lg n} i + \sum_{i=1}^{\lg n} 2^i = \tag{5.2}$$

$$= O(1) + ... + O(1) + 2 + 4 + ... + \frac{n}{2} + n = O(n) \tag{5.3}$$

A visual representation of the algorithm in form of a *directed acyclic graph* and pseudocode can be seen in Figure 5.3 and Algorithm 5.2.4 respectively.

Implementation details

Implementing the algorithm as a sequential and parallel versions for both *Cilk++* and *EAD++* was once again straightforward task. The sequential an parallel version of the code can be seen in Appendix A.1.

An *in place* merge algorithm is already implemented in *C++ STL* that ensures a worst-case time complexity of $O(n \lg n)$. In the initial experiments showed that the *C++ STL inplace merge* outperformed the implementation of the `Merge` algorithm. The experiments also showed that the parallel algorithm did fulfil the minimal requirement of being faster than the best sequential algorithm, *C++ STL sort*. The result of the *practical efficiency measures* based on *speedup*, *efficiency*, and *scalability* can be seen in Chapter 8.

Algorithm 5.2.4: `ParallelMergesort`(A, p, r)

```
Data: Let $A$ be an array.
Data: Let $p$ and $r$ be indexes in the array $A$ such that $p \leq r$.
1 if $p < r$ then
2     $q \leftarrow \frac{\lfloor p+r \rfloor}{2}$;
3     parallel
4         ParallelMergesort($A, p, q$);
5         ParallelMergesort($A, q+1, r$);
6     sync;
7     Merge($A, p, q, r$);
```

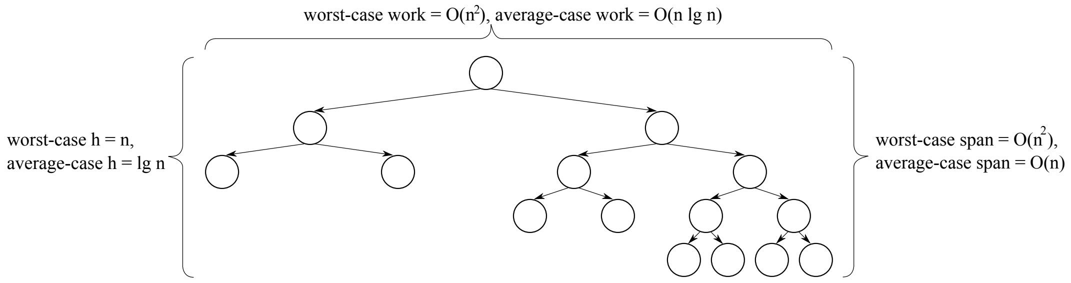


Figure 5.4: Representation of `Quicksort` computation as a DAG.

Algorithm 5.3.1: `Pivot`(A, p, r)

```
Data: Let $A$ be an array.
Data: Let $p$ and $r$ be indexes in the array $A$ such that $p \leq r$.
1 return Random($p, r$);
```

5.3 Quicksort

`Quicksort`, described in Algorithm 5.3.3, is also a comparison-based sorting algorithm, that sorts input sequences of n elements with a worst-case of $O(n^2)$ comparisons but an average of $O(n \lg n)$ comparisons. This algorithm is based on the *divide-and-conquer* paradigm as `Mergesort`. The algorithm is split up into two subsequences. The `Pivot` is chosen randomly, in Algorithm 5.3.1, and based on this the `Partition`, in Algorithm 5.3.2, can create subsequences of different size unbalancing the recursion tree. The algorithm calls itself until the size of the subsequence is one. Compared to `Mergesort`, no bottom-up merging needs to be done. The time complexity of the algorithm is in $O(n^2)$, *worst-case* scenario with $O(n^2)$ comparisons. The average

Algorithm 5.3.2: `Partition`(A, p, q, r)

```
Data: Let $A$ be an array.
Data: Let $p$ and $r$ be indexes in the array $A$ such that $p \leq q \leq r$.
$x \leftarrow A[q]$;
$A[p] \leftrightarrow A[q]$;
$i \leftarrow p$;
for $j \leftarrow p + 1$ to $r$ do
    if $A[j] \leq x$ then
        $i \leftarrow i + 1$;
        $A[i] \leftrightarrow A[j]$;
$A[p] \leftrightarrow A[i + 1]$;
return $i + 1$;
```

time complexity is in $O(n \lg n)$ with $O(n \lg n)$ comparisons. Minimizing the *worst-case* time complexity can be achieved by always setting the `Pivot` to the *median* element in the sequence as in [8]. The median can be found in *linear time*. This will ensure that `Partition` will always split up in two equal sized sequences and the depth of the recursive tree would be bounded to $O(\lg n)$.

Algorithm 5.3.3: `Quicksort`(A, p, r)

```
Data: Let $A$ be an array.
Data: Let $p$ and $r$ be indexes in the array $A$ such that $p \leq r$.
if $p < r$ then
    $q \leftarrow$ Pivot$(A, p, r)$;
    $q' \leftarrow$ Partition$(A, p, q, r)$;
    Quicksort$(A, p, q' - 1)$;
    Quicksort$(A, q' + 1, r)$;
```

The nature of `Quicksort` is also a *divide-and-conquer* algorithm, as `Mergesort`. The same approach can be used in order to parallelize the algorithm. An advantage over `Mergesort` is that `Quicksort` do not need barriers. Once a partition have split the sequence up in two halves there would be no need for merging them back together.

The *work* is $O(n^2)$ for the *worst-case* and $O(n \lg n)$ for the *average-case*. Respectively the *span* is $O(n^2)$ for the *worst-case* and $O(n)$ for the *average-case*. The *critical path* for the *worst-case* is bounded to the partition and the unbalanced recursion tree:

$$T_k(n) = \sum_{i=0}^{n-1} n - i = \tag{5.4}$$

$$= n + (n-1) + (n-2) + ... + 2 = O(n^2) \tag{5.5}$$

The *critical path* for the *average-case* is bounded to the first partition:

$$T_k(n) = \sum_{i=1}^{\lg n} \frac{n}{i} = \tag{5.6}$$

$$= n + \frac{n}{2} + \frac{n}{4} + ... + 2 = O(n) \tag{5.7}$$

A visual representation of the algorithm in form of a *directed acyclic graph* and pseudocode can be seen in Figure 5.4 and Algorithm 5.3.4 respectively.

Algorithm 5.3.4: `ParallelQuicksort`(A, p, r)

```
Data: Let $A$ be an array.
Data: Let $p$ and $r$ be indexes in the array $A$ such that $p \leq r$.
1 if $p < r$ then
2     $q \leftarrow$ Pivot($A, p, r$);
3     $q' \leftarrow$ Partition($A, p, q, r$);
4     parallel
5         ParallelQuicksort($A, p, q' - 1$);
6         ParallelQuicksort($A, q' + 1, r$);
```

Implementation details

Implementing the algorithm as a sequential and parallel versions for both *Cilk++* and *EAD++* was once again straightforward task. The sequential and parallel version of the code can be seen in Appendix A.1.

The *C++ standard template library* offers a function, n^{th} *element*, that ensures that a chosen element in a sequence will be in the same position as if the sequence was sorted, $\langle s_0 \leq \cdots \leq s_n \geq \cdots \geq s_m \rangle$ where $0 \leq n \leq m$. The function can be used to select the *median* of a sequence as the pivot and ensuring that partition always will split the sequence in equal sized subsequences. This will ensure that the recursion tree will be balanced. The time complexity of the function is $O(n)$.

The initial experiments showed that the *C++ STL* n^{th} *element* was slower than the `pivot-partition` combined with the *load-balancing* from *Cilk++* or *EAD++*. The experiments also showed that the parallel algorithm fulfilled the minimal requirement of being faster than the best sequential algorithm. Hereby there will be made *practical efficiency measures* based on *speedup*, *efficiency*, and *scalability*. The results can be seen in Chapter 8.

Chapter 6

Finding minimum spanning trees in parallel

As in the previous chapter, the algorithms for *finding minimum spanning tree* will also be studied with emphasis on how to they can be parallelized. The *worst-case* asymptotic time with minimal extra space utilization and a minimal requirement of being at least as fast as the best sequential algorithm will also be the main criteria to accept or reject an algorithm. The best sequential algorithm will be chosen from the studied algorithms because no algorithm to find minimum spanning trees is part of the *C++ STL*. The algorithms are: `Prim`, `Kruskal` with the `Kruskal-filter` variant and `Borůvka`. The chosen algorithms have in common that they can be implemented on a *Exclusive-Read Exclusive-Write (EREW) parallel random access machine (PRAM)*.

In a *connected and undirected graph*, $G(V, E)$, where V is a set of vertices and E is a set of edges, a *spanning tree* is defined as a tree containing all *vertices* connected through *edges*. These edges must not form cycles. The *minimum spanning tree* is a spanning tree containing all the vertices but only with the minimal amount of edges connecting the vertices.

6.1 Prim (Dijkstra-Jarník-Prim, DJP)

`Prim`, described in Algorithm 6.1.1, finds the *minimum spanning tree*, T, in a *connected and undirected graph*, $G(V, E)$, by:

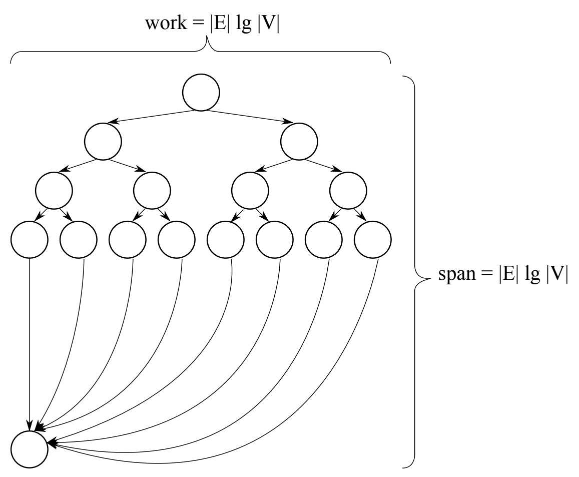


Figure 6.1: Representation of `Prim` computation as a DAG.

- initially setting all the *key* values to ∞ and the closest neighbor, π, to *none* for all the vertices. These operations are made in *linear time*, $O(|V|)$;
- secondly selecting a random vertex, r, and setting the key value to 0. The operation are made in *constant time*, $O(1)$;
- afterwards adding all the vertices to a *min-heap*; The operation is made in *linear time*, $O(|V|)$;
- selecting the vertex with the lowest key value removing it from the heap. In this case the random vertex, r, with a key value of 0. The removed vertex now updates, all its adjacent vertices key values with the weight of the edge from v to u and the closest neighbour, π, with v if the weight of the edge is less than u key value, if they are in the heap. These operations are made in *linear time*, $O(|E| + lg|V|)$;
- repeating the previous operation until there are no more vertices in the heap. These operations are made in *log-linear time*, $O((|E| + |V|) \lg |V|)$;
- iterating through each vertex, except the randomly picked vertex, r, and adding an edge from the vertex to its closest neighbour with a weight of its key value. The operation are made in *linear time*, $O(|V|)$;

Depending on the data structures used to implement the algorithm the time complexity varies from:

Algorithm 6.1.1: `Prim(`G`)`

```
Data: Let $V$ be a set of vertices.
Data: Let $E$ be a set of edges.
Data: Let $G$ be a connected undirected graph $G = (V, E)$.
Data: Let $w(u, v)$ be the weight function of an edge that connects $u$ to $v$.
Data: Let $v_k$ be the minimum weight of any edge connecting $v$ to the tree.
Data: Let $v_\pi$ be the parent vertex of $v$ in the tree.
Data: Let $v_{adj}$ be a set of vertices that $v$ is adjacent to.
Data: Let $H$ be a minimum heap.
Data: Let $T$ be a subgraph of $G$ where $T = \emptyset$.
foreach $v \in V$ do
    $v_k \leftarrow \infty$;
    $v_\pi \leftarrow$ NIL;
$r \leftarrow$ Random($V$);
$r_k \leftarrow 0$;
$H \leftarrow$ Make-heap($V$);
while $H \neq \emptyset$ do
    $u \leftarrow$ Extract-heap($H$);
    foreach $v \in u_{adj}$ do
        if $v \in Q$ and $w(u, v) < v_k$ then
            $v_\pi \leftarrow u$;
            $v_k \leftarrow w(u, v)$;
            Decrease-heap($H, v$);
foreach $v \in V \backslash \{r\}$ do
    $T \leftarrow T \cup (v, v_\pi)$;
return $T$;
```

- $O(|V|^2)$ if an *adjacency-matrix* is used, where $n = |V|$;
- $O((|V| + |E|) \lg |V|) = O(|E| \lg |V|)$ if an *adjacency-list* is used in combination with a *binary heap*;
- $O(|E| + |V| \lg |V|)$ if an *adjacency-list* is used in combination with a *Fibonacci heap*;

A space inconvenience with this algorithm with respect to others *minimum-spanning-tree* algorithms is that it needs to store the neighbors of each vertex in either an *adjacency list or matrix.* The *adjacency matrix* will always need $|V|^2$ extra space and the worst-case scenario for the *adjacency list* would be a *complete graph*, every vertex is connected to all the other vertices, where the *adjacency list* would need $\frac{|V|(|V|-1)}{2}$ extra space.

The main problem in order to parallelize this algorithm is to handle with the *minimum-heap*. While one *thread* performs the operations from line *8-13* in Algorithm 6.1.1. No other thread can have access to the data structure. Once the tread terminates it can perform once again the same operations without needing other threads. Attempts of replacing the heap with other data structures have been tried but without success. The *work* is $O(|E| \lg |V|)$ and the span is $O(|E| \lg |V|)$. The *critical path* is bounded to the sequential operations of the heap combined with the updates on the adjacency lists:

$$\begin{aligned} T_k(n) &= O(|V|) + O(1) + O(|V|) + O((|E| + |V|) \lg |V|) + O(|V|) && (6.1)\\ &= O(|E| \lg |V|) && (6.2) \end{aligned}$$

Since the algorithm will not gain an increased performance when adding more processes, the algorithm is rejected and there will not be performed initial experiments. A visual representation in form of a *directed acyclic graph* can be seen in Figure 6.1.

Implementation details

The algorithm is only implemented in a sequential version and can be seen in Appendix A.1. A *vertex'* class is introduced expanding the initial *vertex* class with the following properties: *key value*, *adjacency list*, *closest neighbor (π)* and *in heap*. There is also implemented a *crease* function that can be used for *decreasing* a node in a minimum heap or *increasing* a node in a maximum heap. The *crease* function complies with the *C++ STL container* policy.

6.2 Kruskal

`Kruskal` algorithm, in Algorithm 6.2.1, finds the *minimum spanning tree*, T, in a *connected and undirected graph*, $G(V, E)$, by:

- initially adding each vertex to its own set of vertices. These operations are made in *linear time*, $O(|V|)$;
- sorting all edges in a non decreasing order by their *weight*. The sorting operation is made in *log-linear time*, $O(|E| \lg |E|)$;

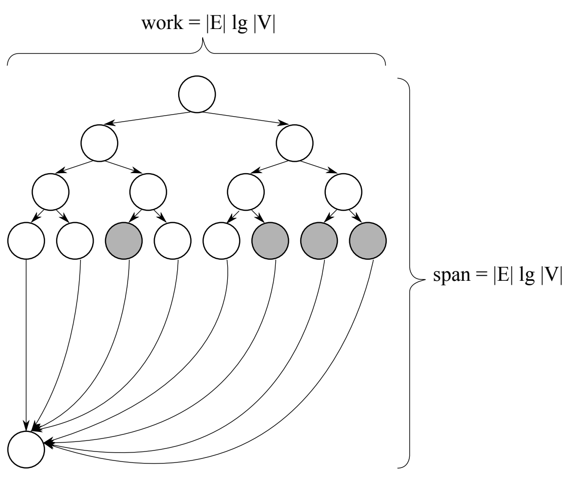


Figure 6.2: Representation of `Kruskal` computation as a DAG.

- selecting each edge in a non decreasing order by their weight and checking if the vertices forming the edge, (u, v), are in the same set of vertices. If they are not, the edge will be added to the minimum spanning tree, T, and the sets will be combined. These operations are made in *log-linear time*, $O((|E| + |V|) \lg |V|)$;

The time complexity is set by the sorting operation, $O(|E| \lg |E|)$. The graph is connected therefore can there only be at most $|E| \leq \frac{|V|(|V|-1)}{2} < |V|^2$ edges. The time complexity is $O(|E| \lg |E|) = O(|E| \lg |V|^2) = O(|E| \lg |V|)$.

The problem with the algorithm is when a graph is *dense*, each vertex has many neighbor vertices, and all the edges have to be sorted. To optimize the algorithm could be to break out of the loop if there only was one set of vertices. This will ensure that all the vertex in the remaining edges would already be in the minimum spanning tree. The problem is that the sorting algorithm already sorted all the edges in $O(|E| \lg |E|)$ and iterating through all the edges can be done in linear time $O(|E|)$. A possibility could be introducing a heap that can be made in $O(|E|)$ time and even though all elements are removed, in case a heavy weighted edge is in the minimum spanning tree, the combined time complexity would still be in $O(|E| \lg |E|)$. Another approach is to use a filter [22] that will exclude all heavy edges before checking for a valid edge. The algorithm is similar to `Quicksort`. It starts by choosing a random edge as pivot and it partition the edges into two list, the first containing all the edges were their weights are less or equal to the pivot edge weight and the other list containing all the edges were their weights greater than the pivot edge weight. The algorithm is called recursively on

```
Algorithm 6.2.1: Kruskal($G$)
Data: Let $V$ be a set of vertices.
Data: Let $E$ be a set of edges.
Data: Let $G$ be a connected undirected graph $G = (V, E)$.
Data: Let $w$ be the weight of an edge.
Data: Let $T$ be a subgraph of $G$ where $T = \emptyset$.
1  foreach $v \in V$ do
2      $V_v = \{v\}$ ;        /* create a subset of $V$ containing only $v$ */
3  Sort($E$);                        /* in a non-decreasing order by $w$ */
4  foreach $(u, v) \in E$ do         /* in a non-decreasing order by $w$ */
5      $V_i \leftarrow$ Find-set($u$);
6      $V_j \leftarrow$ Find-set($v$);
7      if $V_i \neq V_j$ then
8          $T \leftarrow T \cup (u, v)$;
9          $V_k \leftarrow$ Union-set($V_i, V_j$);
10         if $V = V_k$ then
11             break;
12 return $T$;
```

the first list until a single edge is reached. The edge will be added to the minimum spanning tree, T, if the vertices forming the edge, (u, v), are not in the same set of vertices. Once the first part is returned from the recursion, the filter will now be applied on the second list by checking if the vertex, (u, v), in each edge are in the same set of vertices. If this is the case, the edge can safely be removed because both vertex are already part of the minimum spanning tree. Pseudocode for the algorithm can be seen in Algorithms 6.2.3 and 6.2.2.

When parallelizing the algorithm usage of a heap must be avoided, as in the previous section where the `Prim` algorithm was not parallelizable because it uses heap as the primary data structure. By looking at the `Kruskal-filter` algorithm there are really no parallelizable parts. The `Quicksort` is used but it cannot run both subtrees of the recursion concurrently. The last one has to wait until the first one returns and then it can apply the filter on the second branch of the recursion tree. In [22] the parallel sorting algorithm is called whenever the *threshold* is reached. The interesting part would be to know the precise value of the *threshold.* The performance test of the sorting algorithms from the previous chapter, shows that *MCSTL parallel sort* only performs better than the best sequential *STL sort* when the number of elements are

Algorithm 6.2.2: `Kruskal-filter-helper(`E,T`)`

```
Data: Let $V$ be a set of vertices.
Data: Let $E$ be a set of edges.
Data: Let $w$ be the weight of an edge.
Data: Let $T$ be a subgraph of $G$.
1  if $|E| < 2$ then
2      if $E = \{(u,v)\}$ then
3          $V_i \leftarrow$ Find-set($u$);
4          $V_j \leftarrow$ Find-set($v$);
5          if $V_i \neq V_j$ then
6              $T \leftarrow T \cup (u,v)$;
7              Union-set($V_i, V_j$);
8      return;
9  $q \leftarrow$ Pivot($E, p, r$);
10 $q' \leftarrow$ Partition($E, p, q, r$);    /* in a non-decreasing order by $w$ */
11 $E_1 \leftarrow E\{e_p, \ldots, e_{q'-1}\}$;
12 $E_2 \leftarrow E\{e_{q'}, \ldots, e_r\}$;
13 Kruskal-filter-helper($E_1, T$);
14 foreach $(u,v) \in E_2$ do
15     $V_i \leftarrow$ Find-set($u$);
16     $V_j \leftarrow$ Find-set($v$);
17     if $V_i = V_j$ then
18         $E_2 \leftarrow E_2 \backslash (u,v)$;
19 Kruskal-filter-helper($E_2, T$);
```

Algorithm 6.2.3: `Kruskal-filter(`G`)`

```
Data: Let $V$ be a set of vertices.
Data: Let $E$ be a set of edges.
Data: Let $G$ be a connected undirected graph $G = (V, E)$.
Data: Let $T$ be a a subgraph of $G$ where $T = \emptyset$.
1  foreach $v \in V$ do
2      $V_v = \{v\}$ ;        /* create a subset of $V$ containing only $v$ */
3  Kruskal-filter-helper($E, T$);
4  return $T$;
```

greater than 2^{13}. Why not sort all the edges with *MCSTL parallel sort*? The performance test of the sorting algorithms, showed a speedup of more than *20*

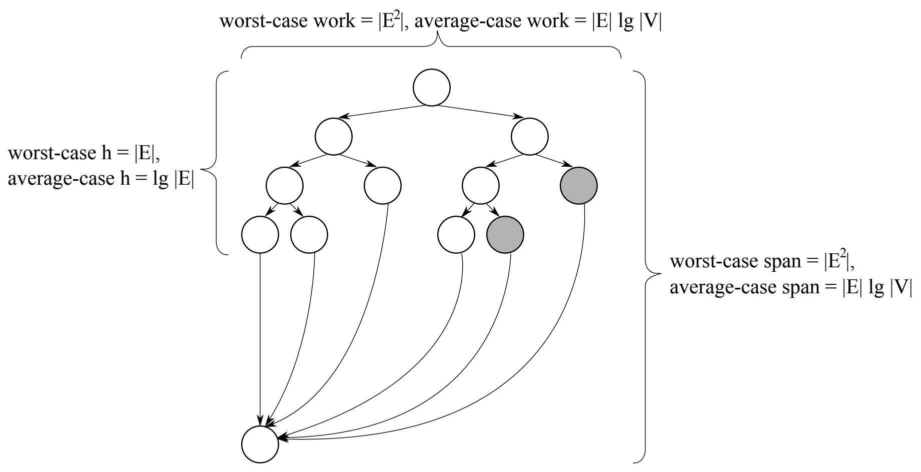

worst-case work = |E2|, average-case work = |E| lg |V|
worst-case h = |E|,
average-case h = lg |E|
worst-case span = |E2|,
average-case span = |E| lg |V|

Figure 6.3: Representation of `Kruskal-filter` computation as a DAG.

when using a computer with *32* kernels compared to the speedup of less than *2* using *MCSTL quicksort*, assuming that `Kruskal-filter` is implemented in the same way as the *MCSTL quicksort* algorithm. More information about performance tests on the sorting algorithms can be seen in Chapter 8.

The *work* for the `Kruskal` algorithm is $O(|E| \lg |V|)$ and the *span* is $O(|E| \lg |V|)$. The *critical path* is bounded by the set operations because a parallel sorting algorithm can be used in order to the reduce the subpath for the sort of the edges from $O(|E| \lg |E|)$ to $O(|E|)$:

$$
\begin{aligned}
T_k(n) &= \sum_{i=1}^{\lg |E|} \frac{|E|}{i} + O((|E| + |V|) \lg |V|) && (6.3)\\
&= |E| + \frac{|E|}{2} + \frac{|E|}{4} + ... + 2 + O(|E| \lg |V|) && (6.4)\\
&= O(|E| \lg |V|) && (6.5)
\end{aligned}
$$

A visual representation in form a *directed acyclic graph* can be seen in Figure 6.2.

On the other hand the *work* for the `Kruskal-filter` algorithms is $O(|E|^2)$ for the *worst-case* and $O(|E| \lg |V|)$ for the *average-case*. Respectively the *span* is $O(|E|^2)$ for the *worst-case* and $O(|E|)$ for the *average-case*. The *critical path* for the *worst-case* is bounded to the partition and the unbalanced recursion tree as in `Quicksort`:

$$
\begin{aligned}
T_k(n) &= \sum_{i=0}^{|E|-1} |E| - i + O((|E| + |V|) \lg |V|) && (6.6)\\
&= |E| + (|E| - 1) + (|E| - 2) + ... + 2 + O(|E| \lg |V|) && (6.7)\\
&= O(|E|^2) && (6.8)
\end{aligned}
$$

The *critical path* for the *average-case* is bounded bounded by the set operations:

$$
\begin{aligned}
T_k(n) &= \sum_{i=1}^{\lg |E|} \frac{|E|}{i} + O((|E| + |V|) \lg |V|) = && (6.9)\\
&= |E| + \frac{|E|}{2} + \frac{|E|}{4} + ... + 2 + O(|E| \lg |V|) && (6.10)\\
&= O(|E| \lg |V|) && (6.11)
\end{aligned}
$$

A visual representation in form a *directed acyclic graph* can be seen in Figure 6.3

Implementation details

Both algorithms are implemented as sequential version but only `Kruskal` is implemented in a parallel version by replacing *C++ STL sort* with *MCSTL parallel sort* and can be seen in Appendix A.1. The set operations rely on *C++ STL set* container.

6.3 Borůvka (Sollin)

Borůvkas algorithm, in Algorithm 6.2.1, works in a similar manner as the `Kruskal` algorithm and finds the *minimum spanning tree*, T, in a *connected and undirected graph*, $G(V, E)$, by:

- initially creating a *supervertex* for each of the vertex. Each supervertex contain a local minimum spanning tree, s_T, a set of adjacent edges to the supervertex, s_{adj}, and all the vertices contained in the supervertex, $s_{vertices}$. These operations are made in *linear time*, $O(|E| + |V|)$;

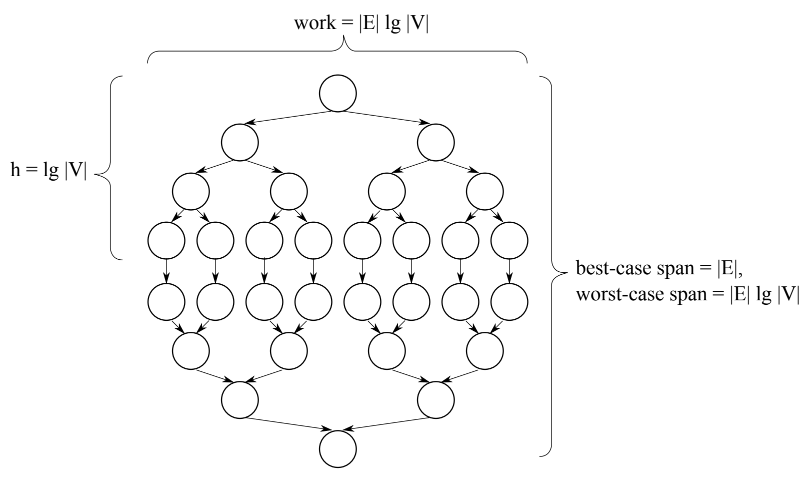


Figure 6.4: Representation of `Borůvka` computation as a DAG.

- taking for each of the supervertices set of adjacent edges, s_{adj}, the edge with the lowest weight. The two supervertices forming the edge will be combined into a new supervertex performing a union on the local minimum spanning trees, s_T, the sets of adjacent edges, s_{adj}, and and list of vertices contained in each of the supervertex, $s_{vertices}$. These operations are made in *linear time*, $O(|E|)$;
- afterwards performing a contraction on the set of adjacent edges, s_{adj}. in order to find the next lowest adjacency edge to another supervertex. The operation is made in *linear time*, $O(|E|)$;
- repeating the previous two operations until there is more than one supervertex. These operations are made in *log-linear time*, $O(|E| \lg |V|)$;

The algorithm can also be thought of as a *bottom-up mergesort* starting with $|V|$ supervertices and *merging* them together ending up with one final supervertex containing all the vertices and the final local minimum spanning tree. The time complexity of the the algorithm is $O(|E| \lg |V|)$.

Because of the similarity to the mergesort algorithm, the algorithm should be parallelizable. The main problem with respect to the merge operation in `Mergesort`, where no other process will access two sublist while they are begin merged, is that several supervertices could try to combine with a common supervertex. In order to ensure that only two supervertices are combined at a given time a lock mechanism needs to be introduced. This approach will create a even bigger problem. Assuming that there exists a graph where all the edges are connected to only one common edge. In order to combine all

Algorithm 6.3.1: `Borůvka(`G`)`

```
Data: Let $V$ be a set of vertices.
Data: Let $E$ be a set of edges.
Data: Let $G$ be a connected undirected graph $G = (V, E)$.
Data: Let $w$ be the weight of an edge.
Data: Let $T$ be a subgraph of $G$ where $T = \emptyset$.
Data: Let $S$ be a set of supervertices where $S = \emptyset$.
Data: Let $s_T$ be a local minimum spanning tree.
Data: Let $s_{adj}$ be a set of edges that $s$ is adjacent to.
Data: Let $s_{vertices}$ be a set of vertices contained in $s$.
foreach $v \in V$ do                    /* convert each $v$ to a supervertex */
    $S \leftarrow S \cup v$;
while $|S| > 1$ do
    foreach $s \in S$ do
        $(u, v) \in s_{adj}$;             /* select the edge with the lowest $w$ */
        $S_i \leftarrow$ Find-supervertex($u$);
        $S_j \leftarrow$ Find-supervertex($v$);
        if $S_i \neq S_j$ then
            $s_T \leftarrow s_T \cup (u, v)$;
            Union-supervertex($S_i, S_j$);
            foreach $(u', v') \in s_{adj}$ do
                $S_{i'} \leftarrow$ Find-supervertex($u'$);
                $S_{j'} \leftarrow$ Find-supervertex($v'$);
                if $S_{i'} = S_{j'}$ then
                    $s_{adj} \leftarrow s_{adj} \backslash (u', v')$;
$s \in S$;
return $s_T$;
```

the supervertices into a single vertex all the computations have to be done sequentially. As in `Prim` with the heap data structure as bottleneck, the algorithm will not gain a better performance when adding more processes.

The *work* for the algorithm is $O(|E| \lg |V|)$ for both the *worst-case* and *best-case*. Respectively the *span* for the *worst-case* is $O(|E| \lg |V|)$ and $O(|E|)$ for the *best-case*. The *critical path* for the *worst-case* is bounded to the lock mechanism on the merging of the supervertices:

$$\begin{aligned} T_k(n) &= O(|E| + |V|) + O(E) + O(E) + O(|E| \lg |V|) && (6.12) \\ &= O(|E| \lg |V|) && (6.13) \end{aligned}$$

The *critical path* for the *best-case* is:

$$\begin{aligned} T_k(n) &= O(|E| + |V|) + O(E) + O(E) + \sum_{i=1}^{\lg |V|} \frac{|E|}{i} && (6.14) \\ &= O(|E|) + O(|E|) + O(|E|) + |E| + \frac{|E|}{2} + \frac{|E|}{4} + ... + 2 && (6.15) \\ &= O(|E|) && (6.16) \end{aligned}$$

Because the studied algorithms are accepted or rejected on basis of their *worst-case* or *average-case* time complexities this algorithm is rejected, as `Prim` was, and there will not be performed initial experiments. A visual representation in form a *directed acyclic graph* can be seen in Figure 6.4.

Implementation details

The algorithm is only implemented in a sequential version and can be seen in Appendix A.1. A *supervertex* class is introduced expanding the initial *vertex* class with the following properties: *local minimum spanning tree*, *adjacency list of edges* and *list of contained vertices.*

An implementation of this algorithm exists for the *boost* library, [5, 10]. The *practical efficiency measures*[1] shows an almost *perfect linear speedup* for *dense graphs.*

A *dense graph* is defined as a graph, $G(V, E)$, where V is a set of vertices and E is a set of edges, where the set of edges E are close to the maximum number of edges. The maximum number of edges for a *connected and undirected graph* is $\frac{|V|(|V|-1)}{2}$ which corresponds to a *complete graph.* Taking the values used by *boost* where $|V| = 100000$ and $|E| = 15000000$ the percentage of how dense the graph is, will be calculated:

[1]http://www.boost.org/doc/libs/1_41_0/libs/graph_parallel/doc/html/dehne_gotz_min_spanning_tree.html

By assuming that the graph is complete, then it would have a density of *100%*:

$$100\ \% = \frac{|V|(|V|-1)}{2} = \frac{100000(100000-1)}{2} \tag{6.17}$$

Calculating the density of the graph used by boost is very simple with cross-multiplication:

$$x\ \% = 15000000 \tag{6.18}$$

$$x\ \% = \frac{100 \cdot 15000000}{\frac{100000(100000-1)}{2}} \tag{6.19}$$

$$= \underline{\underline{0.300003\ \%}} \tag{6.20}$$

From (6.20) a conclusion can be made that the used graphs are not dense.

Chapter 7

Experimental setup

In this chapter the different hardware architectures on which the experiments were executed will be described. Afterwards, the used compilers in order to create binaries are mentioned. Finally, how the pseudo-random input data is created for the tests and a guide on how the reader can reproduce both correctness and performance test is presented.

The initial experiments from the *fifth* step, in Figure 2.2, were performed on following hardware:

Single-dualcore:

- Hardware type: Laptop.
- Processor: Single Intel(R) Core 2 Duo 2.60GHz and 6144 KB cache each core.
- Memory: 4 GB RAM.
- Operating system: Mac OS X v.10.6.4 (64-bit).

The hardware was used to reject algorithms that did not fulfill the minimal requirement of being faster than the best sequential algorithm. If an algorithm was not rejected, *practical efficiency measures* based on *speedup*, *efficiency* and *scalability* with real data, the *seventh* step in Figure 2.2, were performed on the following hardware:

Dual-hyperthreading:

- Hardware type: Server.
- Processor: Dual Intel(R) Xeon(TM) 2.80GHz with Hyperthreathing enabled and 512 KB cache in each processor.
- Memory: 2 GB RAM.
- Address: benzbox.********.diku.dk
- Operating system: GNU/Linux Ubuntu 10.04 LTS (32-bit).

Dual-quadcore:

- Hardware type: Server.
- Processor: Dual Intel(R) Quad-Core Xeon(R) X5355 2.66GHz and 4096 KB cache in each core.
- Memory: 16 GB RAM.
- Address: knapsack2.********.diku.dk
- Operating system: GNU/Linux Ubuntu 10.04 LTS (64-bit).

Tetra-octacore:

- Hardware type: Server.
- Processor: Tetra AMD Opteron(tm) Octa-Core Processor 6136 2.40GHz and 512 KB cache in each core.
- Memory: 128 GB RAM.
- Address: octopus.********.unipi.it
- Operating system: SUSE Linux Enterprise Server 11 (64-bit).

The servers were used exclusively. No other processes, besides those from the operating system, were running under the performance tests. It is mentioned, in Section 1.2, that the ideal mapping between thread and cores is usually *1-to-1*. This is confirmed on all tested platforms. Because of the limited time to use the servers, the tests were run only a few times but always with the same outcome. The ideal would be to run the test many times and calculate the average of the results.

The *GNU Compiler Collection (GCC)* was used in order to compile the source code, a minimal requirement of *MCSTL* is *GCC 4.3*. Note that the *Cilk++* code can only be compiled with *Intel's* own version of *GCC*. *EAD++* can also be compiled using *Low Level Virtual Machine (LLVM)* and *Intel C++ Compiler (ICC).*

The input data for the sorting algorithms are pseudo-randomly generated lists of size 2^n. The interval of the pseudo-random numbers are in the range $\left[1, \frac{2^n}{2}\right]$, this will ensure that there will be duplicate numbers. The input data for the *minimum-spanning-tree* algorithms are also pseudo-randomly generated graphs with at most one minimum spanning tree. The initial created list of vertices is *shuffled* and then in linear time each vertex will create an edge with its predecessor vertex and an edge weight in the range $\left[1, \frac{2^n}{2}\right]$ creating a minimum spanning tree containing all vertices. The maximum degree of the graph is set to half the number of vertices. Based on this value several iteration will be made in order to create new edges from random vertices, if they do not exist already, with an edge weight in the range $\left[\frac{2^n}{2} + 1, 2^n\right]$. The random input data is generated before the algorithms are executed.

Correctness tests for the sorting algorithms checks on the output if the next number is greater than the previous. Correctness test for the *minimum-spanning-tree* algorithms are done by checking all edges in the outputted list against the initial vertex list by finding the first vertex from the edge in the vertex list. When the vertex is found, the second vertex from the edge must match the next vertex in the initial vertex list.

The execution of the algorithms are measured in *nanoseconds*. Creation of pseudo-random input data is not included in the time measuring.

Both correctness and benchmark test can be reproduced by the reader in order to confirm the results:

Reproducing correctness tests

From */thesis/src* type *make test*. Go to */thesis/src/test* and type *sort.bash 1 20 1 2* or *mst.bash 1 20 1 2* which stand for execute all the binary files created and give as a parameter $n = 2^1 \ldots 2^{20}$ and $k = 1 \ldots 2$. Where n is the number of elements and k is the number of processors to be used. There will now be created a folder named after date combined with time inside */thesis/src/test/output* containing all the input of the executed applications each in a separated file. By typing *grep -n '0' *.txt*, will show if any algorithm has failed. If all files only contain *1*, the algorithms are tested to work properly. The source code and scripts can be seen in Appendix B.1.

Reproducing benchmark tests

From */thesis/src* type *make benchmark*. Go to */thesis/src/benchmark* and type *sort.bash 1 20 1 2* or *mst.bash 1 20 1 2* which stand for execute all the binary files created and give as a parameter $n = 2^1 \dots 2^{20}$ and $k = 1 \dots 2$. Where n is the number of elements and k is the number of processors to be used. There will now be created a folder named after date combined with time inside */thesis/src/benchmark/output* containing all the input of the executed applications each in a separated file. The source code and scripts can be seen in Appendix B.2.

Chapter 8

Results and discussion

In this chapter the results of the benchmarks for executing the *sorting* and *minimum spanning tree* algorithms on different computer architectures are presented. Only one parallel version of a *minimum-spanning-tree* algorithm was implemented, therefore no *practical efficiency measures* are made for those algorithms. The *practical efficiency measures* for the sorting algorithms are visualized as *plots*. The presented plots are limited to the results of using 2^k, where $k = 2^i,\ i \in \mathbb{N}$, processors. Finally based on the hypothesis and benchmark results, algorithms will be accepted or rejected for the library.

8.1 Sorting

Only `MergesortParallel` and `QuicksortParallel` in their *Cilk++* and *EAD++* implementations were benchmarked. As mentioned in Section 5.1, `Bitonicsort` did not fulfill the minimal requirement of being as fast as the best sequential algorithm. The two algorithms chosen to measure the parallel implementations are the *C++ STL sort*, which is the best sequential sorting algorithm, and *C++ STL parallel sort*, which is *MCSTL's* implementation of *mergesort*.

Dual-hyperthreading server

The initial experiments were executed on this server. The result showed that the *MCSTL* implementation of *mergesort* was much faster than the versions implemented in *Cilk++* and *EAD++*. The *speedup* and *efficiency* is similar to the others implementations until the size of the sequence exceeds 2^{10} as in Figure C.1 and C.3. Note that when the sequence is of size 2^{28} the *speedup* and *efficiency* of the *MCSTL mergesort* is similar to the other two implemented algorithms. This is due to the memory usage of the algorithm. The server only have *2* GB of RAM and it was useful to show what happens when each of the algorithms reach the memory limit. In case of *MCSTL mergesort* the practical performance falls drastically. In order to compare what happened when the algorithms run out of memory the following test case was run by setting the size of the sequence to sort to 2^{29}:

```
ramon@benzbox:~/$ ./sort_par_tp_mergesort-g++ --n=29 --k=4
Segmentation fault
ramon@benzbox:~/$ ./sort_par_cilk++_mergesort-cilk++ --n=29 -cilk_set_worker_count=4
Bus error
ramon@benzbox:~/$ ./sort_par_mcstl_std_parallel_mergesort-g++ --n=29 --k=4
0.000093411
```

EAD++ implementation crashed with a *segmentation fault*, *Cilk++* implementation also crashed with a *bus error*. The only implementation that did not crash was *MCSTL*. Guessing that no algorithm is able to sort a sequence of 2^{29} numbers in less than 0.1 milliseconds, an error message is expected. Specially because this implementation is part of the *C++ STL parallel library*.

The implementations of the `Quicksort` algorithm showed that once again the *MCSTL* version was the fastest. The practical performance decreases drastically when a sequences are greater than 2^{25}. An explanation can be that because the algorithm calls the *C++ STL sort*, a sequential algorithm, whenever there are no more threads available. In case the recursion tree is unbalanced and some threads finish before others. The finished threads cannot help due to the algorithm now runs sequentially. The *MCSTL load-balanced* version of *quicksort*, based on the *work-stealing* technique, is the fastest when *2* threads are used. When *4* threads are used then it is *EAD++* which is the fastest, outperforming even the *C++ STL parallel sort* for 2^{28}. The *Cilk++* implementation has some discrete results. The results can be seen in Figure C.2 and C.4.

The *scalability* plots, from Figure C.5 to C.8, show that the performance for algorithms implemented in *Cilk++* and *EAD++* grow whenever extra processors are added. On the other hand, the performance of the algorithms implemented in *MCSTL* grow fast, *peak*, and then falls. Note on the strange behaviour of the *MCSTL mergesort*. This is produced by the algorithm when reaching the memory limit. The problem is that when executing the algorithm with only one processor, sequentially, on a sequence of of size 2^{28} the execution time is *318.01* seconds. Comparing this time to when the algorithm sorts a sequence of size 2^{27} with one processor in only *23.55* seconds. This would move the time complexity of the algorithm from $O(n \lg n)$ to $O(n^2)$.

All the *speedup* plots (Figures C.1 to C.2), *efficiency* plots (Figures C.3 to C.4) and *scalability* plots (Figures C.5 to C.8) can be seen in Appendix C.1.

Dual-quadcore server

The limitation for the usage of this server only allowed to perform *practical efficiency measures* on sequences in the range of 2^{26} to 2^{30}. The results on this server are very similar to the previous. *MCSTL mergesort* is still the fastest and without the memory barrier there are no drastic falls in the performance. The implementation of `Mergesort` on the other two frameworks, *Cilk++* and *EAD++* are almost equal.

The results on the implementation of `Quicksort` shows that it is now the *Cilk++* version that is the best followed closely by the *EAD++* implementation.

All the *speedup* plots (Figures C.9 to C.12), *efficiency* plots (Figures C.13 to C.16) and *scalability* plots (Figures C.17 to C.20) can be seen in Appendix C.1.

Tetra-octacore server

The result on this server confirms what have been seen until now. The best parallel sorting algorithm implementation is *MCSTL mergesort*. The best `Quicksort` is the version implemented in *Cilk++*, followed very closely by the version from *EAD++*.

All the *speedup* plots (Figures C.21 to C.26), *efficiency* plots (Figures C.27 to C.32) and *scalability* plots (Figures C.33 to C.37) can be seen in Appendix C.1.

8.2 Finding minimum spanning trees

There was only made benchmark test on the *dual-hyperthreading server* and *dual-quadcore server*. Access to the *tetra-octacore server* was no longer available. No *practical efficiency measures* were made because only one parallel algorithm was implemented. The implementation of `Kruskal` showed to be the fastest sequential algorithm. In order to execute the benchmark test and because the server usage was limited by time, the `Borůvka` algorithm was excluded. The initial test showed that the implementation of `Borůvka` was much slower than the others.

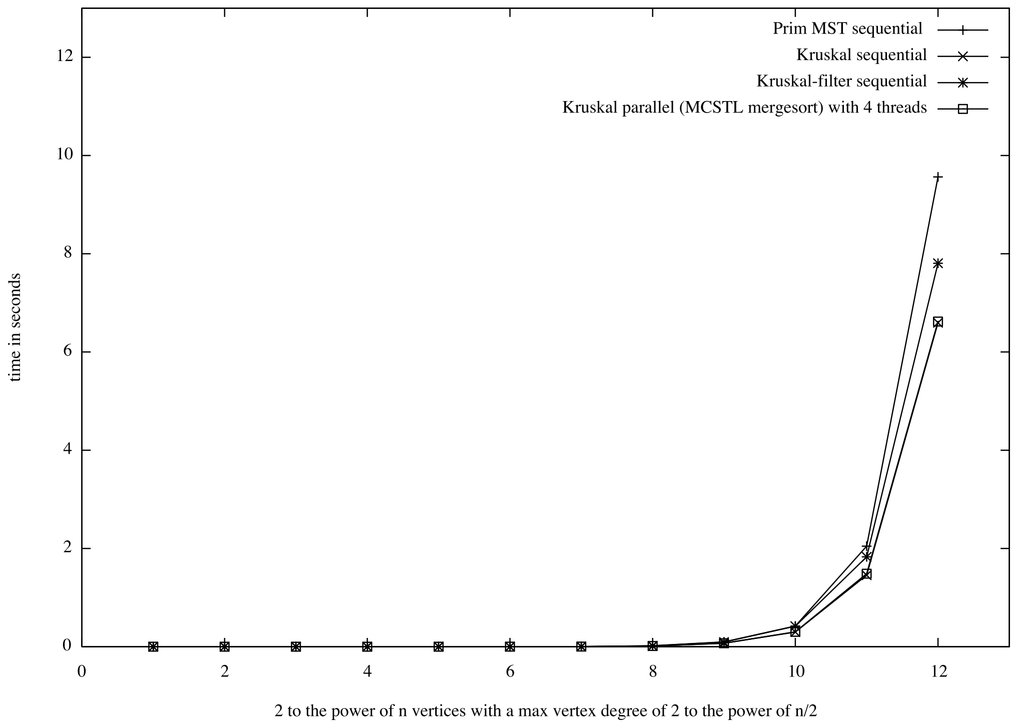


Figure 8.1: MST time on dual-hyperthreading.

The test showed that `Prim` was the slowest sequential algorithm and that `Kruskal` was the fastest. The problem with `Kruskal-filter` is that is is now bounded to the *worst-case* time complexity of `Quicksort`, which is $|E|^2$, while `Kruskal` is still bounded to time complexity of *C++ STL sorts*, $|E| \lg |V|$. The best sequential algorithm to measure against the `Kruskal` parallel implementation based on the *C++ STL parallel sort* is `Kruskal`. As can be seen in Figure 8.1 and 8.2, the results are very close. The fastest is the parallel implementation of `Kruskal` based on the *C++ STL parallel sort*.

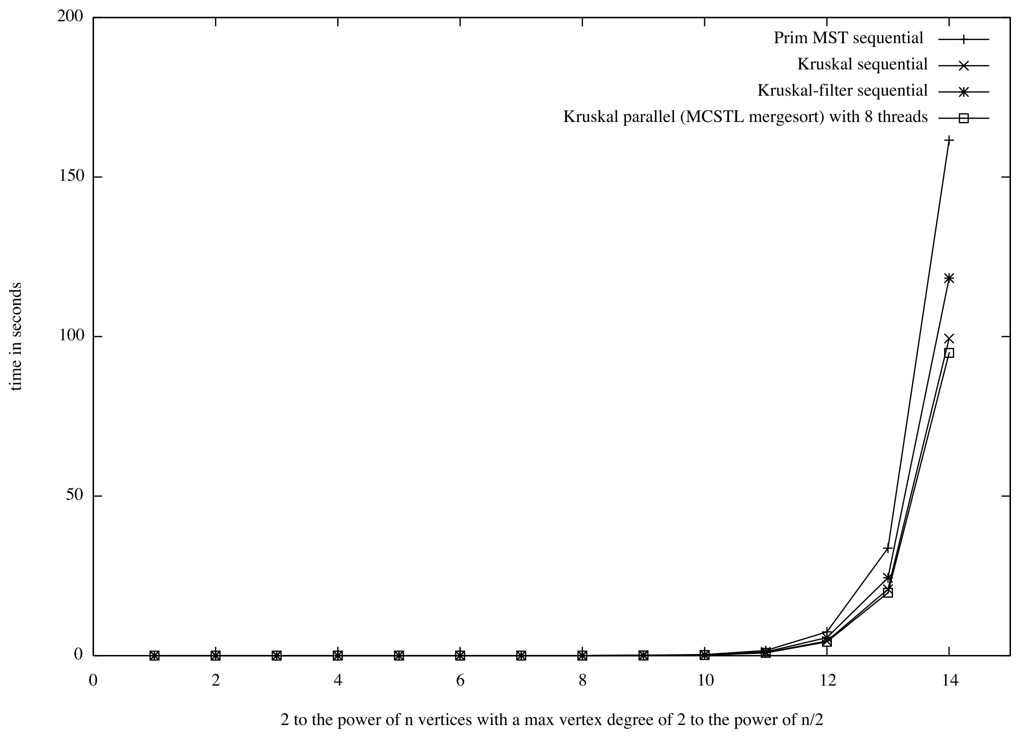


Figure 8.2: MST time on dual-quadcore.

8.3 Discussion

Based on our hypothesis where the minimum requirement is that the algorithm has to be faster than the best sequential algorithm. Both the implementations of `Mergesort` and `Quicksort` for *Cilk++* and *EAD++* are accepted. Likewise is the parallel `Kruskal` implementation based on *C++ STL parallel sort*.

Out from these results what stands out is the implementation of *MCSTL mergesort*. The test performed on the *tetra-octacore* server showed a *speedup* superior to *20*. How is this possible? The answer is that the `Mergesort` algorithm implemented in *MCSTL* is not based on a *binary merge* as the versions implemented for *Cilk++* and *EAD++* but its based on a *multiway merge*. The *multiway mergesort* in [20], is an *external sorting* algorithm. An *external sorting* algorithm is based on a memory model where there exist a fast internal memory of size M and a large external memory. The data is moved between memories in block of size B. How the *multiway mergesort* takes advantage of the internal memory is that it can hold more than one block. Assuming that there can be k blocks, an output block and a small amount of extra storage in the internal memory. Then k blocks containing sorted sequences can be merged in a *k-way*. The merging of the k sequences is done with help of a priority queue containing the smallest element from each sequence. Insertion of elements and extraction of the minimum element of the priority queue can be done in *logarithmic time* because its built on a *minimum-heap*. The smallest element is written to the output block. Once the output block is full. The sequence of sorted elements is written to the *external memory*. If one of the k blocks is empty in the *internal memory* a new block is retrieved from the *external memory*. The bound set in order to determine the size of the k blocks is set to:

$$M \geq (k)B + B + k \tag{8.1}$$

Where the terms are for the k blocks, the output buffer and the priority queue. By doing this the number of merging phases will then be reduced to $\left\lceil \log_k \left(\frac{n}{M}\right) \right\rceil$ and the time complexity of the algorithm becomes:

$$T(n) = O\left(\frac{n}{B}\left(1 + \left\lceil \log_{\frac{M}{B}} \left(\frac{n}{M}\right) \right\rceil\right)\right) \tag{8.2}$$

In the *MCSTL multiway mergesort*, the *external memory* is the shared memory and the *internal memory* is the cache of the *k processors*. The more processors added the more internal memory will be available. By running the algorithm against the sequential version of *mergesort* a *speedup* of *50* was achieved when using *32* processors getting *superlinear speedup*. That is why *multiway mergesort* is chosen to be the *C++ STL parallel sort* as *introsort* was chosen to be the *C++ STL sort*.

Chapter 9

Closure

9.1 Conclusion

The two primary goals of the thesis was: to make an experimental study of the various programming models to implement parallelism from a practical approach and evaluate them with emphasis on efficiency on multi-core computers; and to design and implement a framework that could be used by researchers in future work. Both goals are achieved.

A new terminology is introduced, $T_k(n)$, which stands for: *"The running time of an algorithm on a computer with k processors where k is defined as a bijection of the k processors to the optimal amount of threads that can be executed concurrently on a given hardware processor."*. The terminology helps to avoid unexpected results when performing *practical efficiency measures.*

In the *Efficient Algorithm and Data Structures framework for C++ (EAD++)*, a new method to *load-balance* work between inactive processors is introduced. The *helping technique* can distribute work between inactive processors in *logarithmic time.* Two types of barriers are also introduced: a *global thread* barrier that can be set by the main thread in order to avoid having to add a barrier in each functions that are called, as seen in *pthread* and *Cilk++*; and a *job barriers* that can chose to execute a given job which is binded to the barrier if no other process is performing the task instead of just waiting.

The implemented framework, *EAD++*, shows a similar performance on the studied algorithms if compared to other frameworks such as *Cilk++* or *MC-STL.* The usage of memory is limited to values given on initialization. This would make the framework suitable for *embedded devices* or upcoming *smartphones* with several cores and a limited amount of memory.

As for the studied algorithms, the `Bitonicsorter` will still need more cores in order to perform better as shown in a recent study. The `Mergesort` and `Quicksort` are bounded to *Amdahl's law* by having a computation that takes *linear time*. With regard to the *minimum-spanning-tree* algorithms, a study on how to replace the sequential data structures should be made.

`Kruskal-filter` and `Borůvka` only perform well on certain graphs, since the first one is bounded to a sequential `Quicksort` and the other only performs well for *sparse* graphs. Full results must be made available, specifically testing on *worst-case* scenarios.

A general thought for designing parallel algorithms should be to avoid as much as possible operations that only can be run sequentially. A good example of this is *multiway mergesort* that tackles the sequential *merge* operation of the `Mergesort` algorithm, which restricts the critical path, by executing the process in parallel. Results of *superlinear speedup* are achieved when comparing to the sequential version of the algorithm.

9.2 Future work

Is it possible to make *EAD++* perform better? A likely bottleneck in the implementation could be the job pool, because it is implemented as an *array of pointers*. This is due to the use of *polymorphism*. Two functions in *TR1*, *std::tr1::function*[1] and *std::tr1::bind*[2] can be used to actually introduce *anonymous functions* know from *functional programming language*. This way the job pool could be defined as an array of functions and then the use of the operator *new*, which dynamically allocates memory on the *memory heap* for each object, could be avoided. A draft of how the changes would look in the current code can be seen in Listing 9.1. This will actually simplifies the interface, no need to encapsulate functions in classes. A challenge would be to add properties to the jobs like *priority* and *status* in order to be used for the *job barriers*. These problems are certainly not trivial. Note that proposals for *TR1* are most likely going to be part of the upcoming version of C++ (*C++0x*).

[1]Polymorphic Function Wrappers (TR1 document nr. n1402): http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2002/n1402.html

[2]Function Object Binders (TR1 document nr. n1455): http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1455.htm

```
...
#include <tr1/functional>
...

class ThreadPool{
 ...
protected:
 ...
 vector<function<void ()> > jl;  // job list, default array (LIFO)
 ...
public:
 ...
  void add(function<void ()> & job){ ... }
 ...
};

...
void gtp_add(function<void ()> & job){ ... }
...

template<typename RandomAccessIterator>
void foo(RandomAccessIterator first, RandomAccessIterator last){
  if(first == last){ return; }
  RandomAccessIterator middle(first+((last - first) >> 1));
  gtp_add(bind(foo,first,middle));
  foo(middle,last);
}

template<typename RandomAccessIterator>
void bar(RandomAccessIterator first, RandomAccessIterator last,
         const size_t & nr_threads = 1, const bool & heap = false,
         const size_t & buffer = 0){
  gtp_init(nr_threads, heap, buffer);
  gtp_add(bind(foo<RandomAccessIterator>,first,last));
  gtp_sync_all();
  gtp_stop();
}
```

Listing 9.1: Proposal to avoid use of polymorphism with help of anonymous functions

As for the algorithms. It would be interesting to see how well a implementation of the *multiway mergesort* would perform in *Cilk++* or *EAD++*.

9.3 Framework availability

The *Efficient Algorithm and Data Structures framework for C++ (EAD++)* and source code, published under a *open source license*, will be made available through either the *CPH STL* or a open source repository such as *sourceforge.com*.

Bibliography

[1] G. M. Amdahl, Validity of the single processor approach to achieving large scale computing capabilities, *Proceedings of Spring Joint Lomputer Lonference*, ACM, New York, NY, USA (1967), 483–485.

[2] R. D. Blumofe and C. E. Leiserson, Scheduling multithreaded computations by work stealing, *J. ACM* **46**, 5 (1999), 720–748.

[3] B. Boehm, A. Egyed, J. Kwan, D. Port, A. Shah, and R. Madachy, Using the WinWin Spiral Model: A Case Study, *Computer* **31**, 7 (1998), 33–44.

[4] S. Chen, J. Zhao, J. Qin, Y. Xie, and P.-A. Heng, An Efficient Sorting Algorithm with CUDA, *Journal of the Chinese Institute of Engineers* **32**, 7 (2009), 915–921.

[5] S. Chung and A. Condon, Parallel Implementation of Borůvka's Minimum Spanning Tree Algorithm, *10th International Parallel Processing Symposium*, IEEE Computer Society Press (1996), 302.

[6] P. Ciechanowicz, M. Poldner, and H. Kuchen, The Müenster Skeleton Library Muesli - A Comprehensive Overview, *Working Papers*, 7 (2009).

[7] M. Cole, Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming, *Parallel Computing* **30**, 3 (2004), 389–406.

[8] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3^{rd} Edition, MIT Press (2009).

[9] J. A. Dam, DikuSTM: A Transactional Memory Library for C++ and C++0x, Master's thesis, University of Copenhagen, Department of Computer Science (2010).

[10] F. Dehne and S. Götz, Practical Parallel Algorithms for Minimum Spanning Trees, *Proceedings of the The 17th IEEE Symposium on Reliable Distributed Systems*, IEEE Computer Society Press (1998), 366.

[11] K. Egdø, A Software Transactional Memory Library for C++, Master's thesis, University of Copenhagen, Department of Computer Science (2008).

[12] J. L. Gustafson, Reevaluating Amdahl's law, *Commun. ACM* **31**, 5 (1988), 532–533.

[13] C. A. R. Hoare, Communicating Sequential Processes, *Commun. ACM* **21**, 8 (1978), 666–677.

[14] J. JáJá, *Introduction to Parallel Algorithms*, Addison-Wesley Professional (1992).

[15] J.Singler, P.Sanders, and F. Putze, MCSTL: The multi-core standard template library (poster), *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP)*, ACM (2007).

[16] J. Katajainen, T. Pasanen, and J. Teuhola, Practical in-place mergesort, *Nordic Journal of Computing* **3**, 1 (1996), 27–40.

[17] J. Kider, *GPU as a Parallel Machine: Sorting on the GPU*, Lecture of University of Pennsylvania (2005).

[18] C. E. Leiserson, The Cilk++ concurrency platform, *Proceedings of the 46th Annual Design Automation Conference*, ACM (2009), 522–527.

[19] T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*, 1^{st} Edition, Addison-Wesley Professional (2004).

[20] K. Mehlhorn and P. Sanders, *Algorithms and Data Structures: The Basic Toolbox*, Springer Berlin Heidelberg (2010).

[21] R. Membarth, F. Hannig, J. Teich, M. Körner, and W. Eckert, Comparison of Parallelization Frameworks for Shared Memory Multi-Core Architectures, *Proceedings of the Embedded World Conference*, World Scientific and Engineering Academy and Society (WSEAS), Nuremberg, Germany (2010).

[22] V. Osipov, P. Sanders, and J. Singler, The Filter-Kruskal Minimum Spanning Tree Algorithm, *The Workshop on Algorithm Engineering and Experiments (ALENEX09)*, ACM (2009).

[23] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*, 7^{th} Edition, John Wiley & Sons (2004).

[24] J. Singler, P. Sanders, and F. Putze, MCSTL: The multi-core standard template library, *Proceeding of the 13th International Euro-Par Conference*, *Lecture Notes in Computer Science* **4641**, Springer (2007), 682–694.

[25] B. Stroustrup, *The C++ Programming Language: Special Edition*, 3^{rd} Edition, Addison-Wesley Professional (2000).

[26] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley Professional (2002).

[27] B. Wilkinson and M. Allen, *Parallel Programming: Techniques and Applications Using Networked Workstations and Parallel Computers*, 2^{nd} Edition, Prentice Hall (2004).

Appendix A

Source code

The source code is included a zip file named: *appendices_A_and_B.zip*

A.1 Efficient library for C++

Sorting algorithms

- */thesis/src/seq/sort.hpp*
- */thesis/src/ead/sort.hpp*
- */thesis/src/cilk++/sort.hpp*

Minimum spanning tree algorithms

- */thesis/src/seq/mst.hpp*
- */thesis/src/mcstl/mst.hpp*

A.2 EAD++ framework

/thesis/src/ead++/ead++.hpp

Appendix B

Test

The source code and script are included a zip file named: *appendices_A_and_B.zip*

B.1 Correctness

- */thesis/src/*
- */thesis/src/test/*

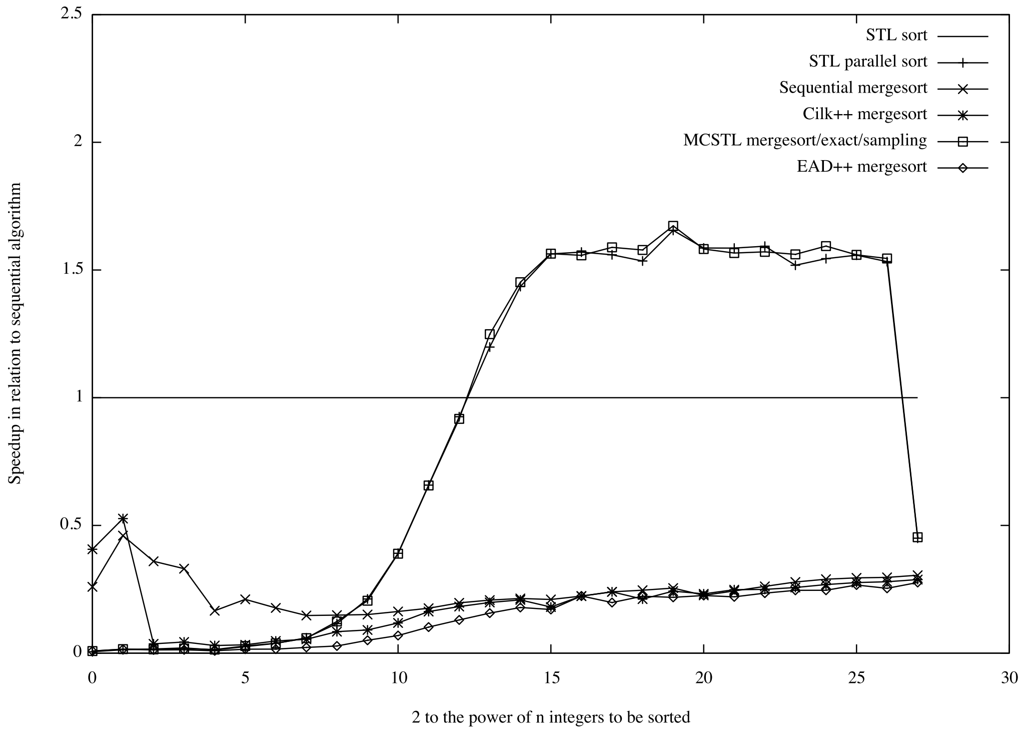
B.2 Benchmark

- */thesis/src/*
- */thesis/src/benchmark/*

Appendix C

Performance measures for sorting algorithms

C.1 On dual-hyperthreading server



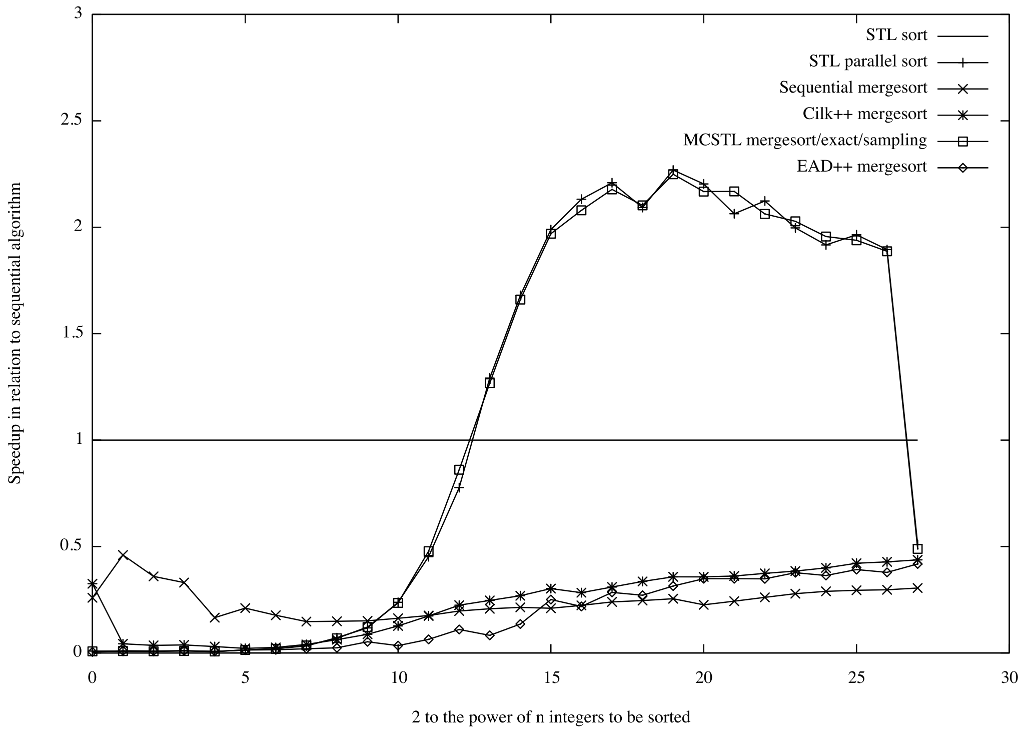
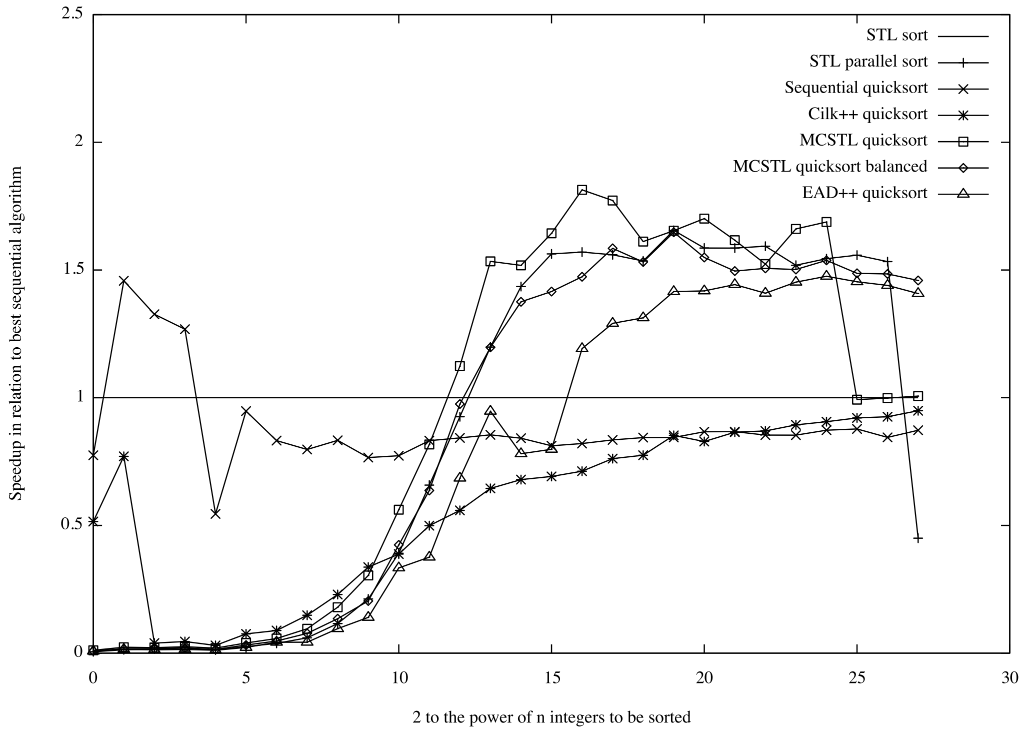


Figure C.1: Mergesort speedup on dual-hyperthreading with 2 and 4 threads.



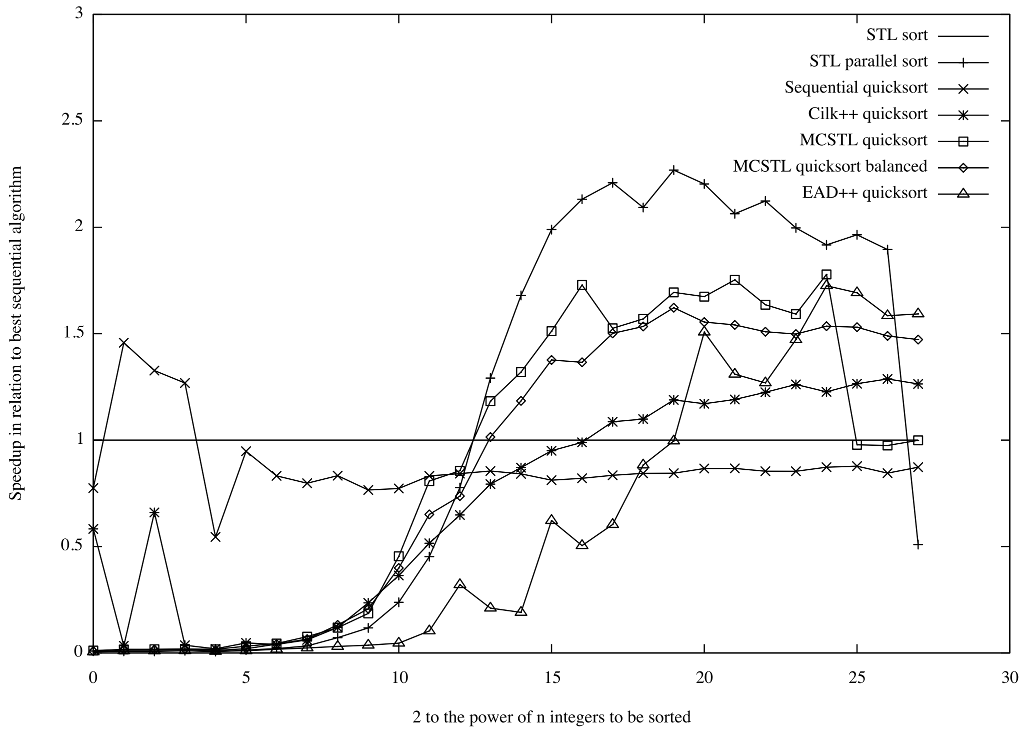
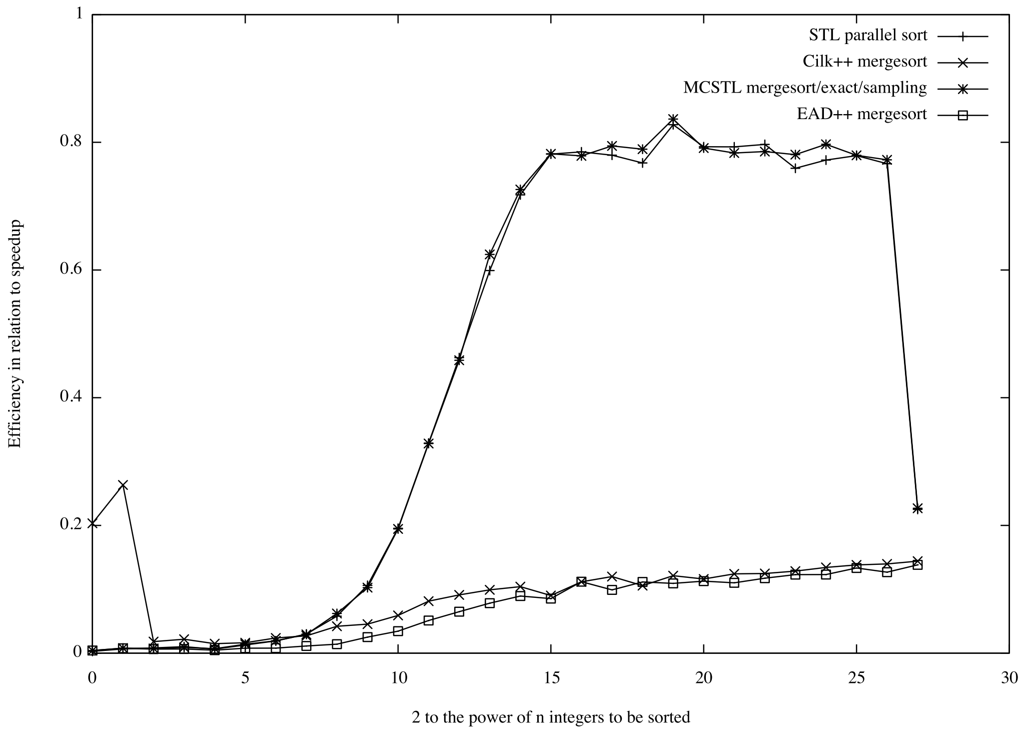


Figure C.2: Quicksort speedup on dual-hyperthreading with 2 and 4 threads.



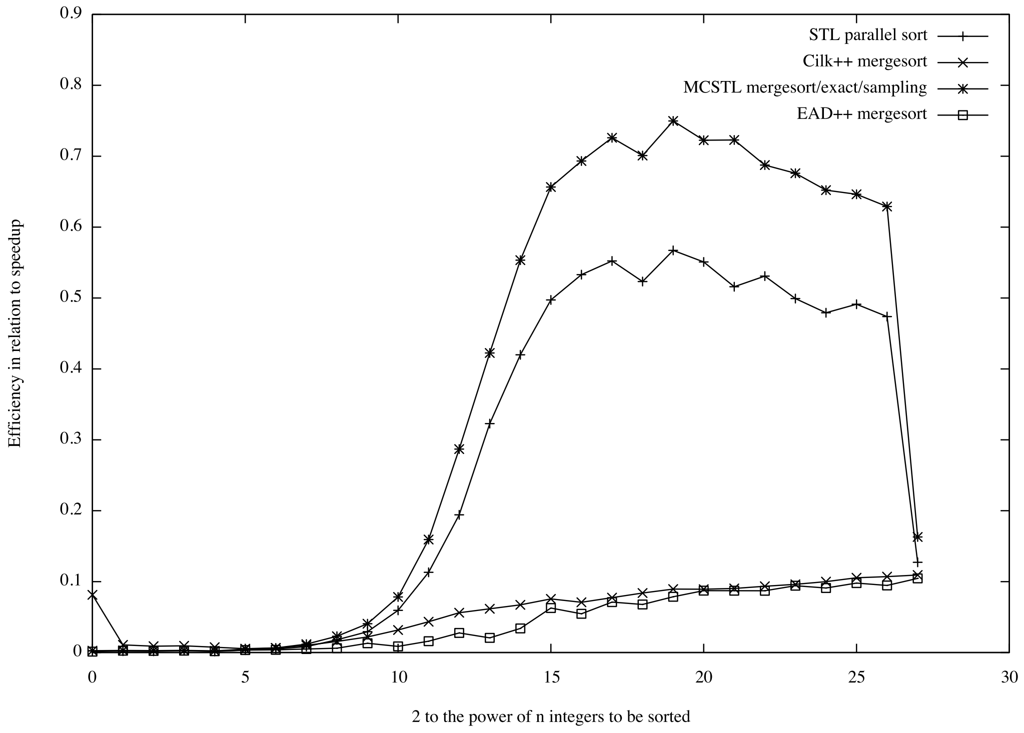
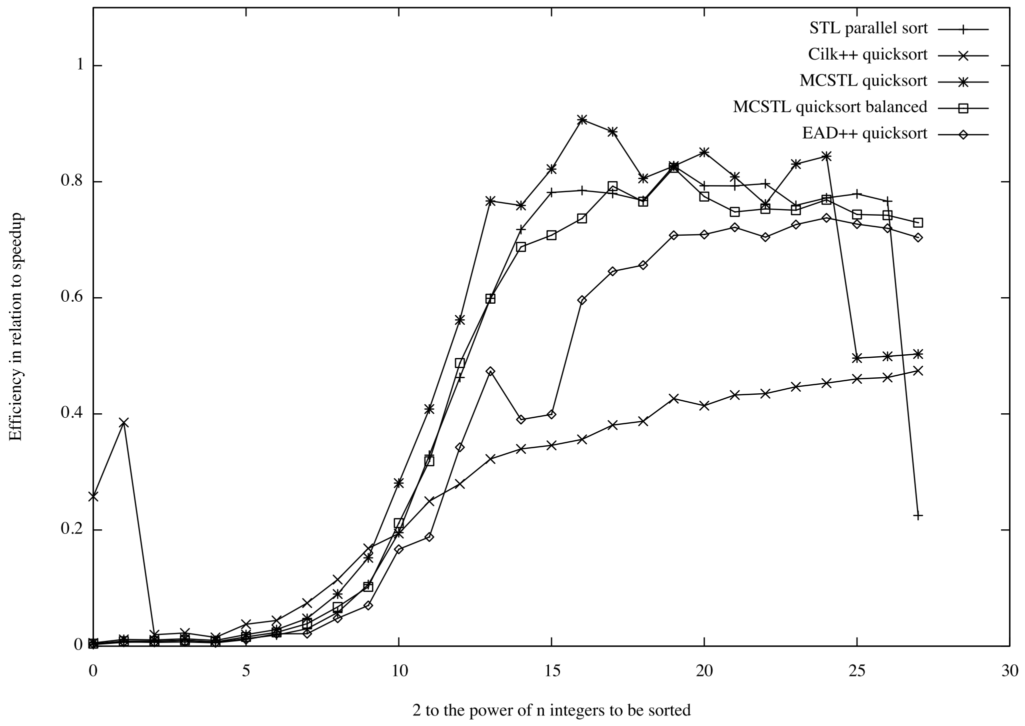


Figure C.3: Mergesort efficiency on dual-hyperthreading with 2 and 4 threads.



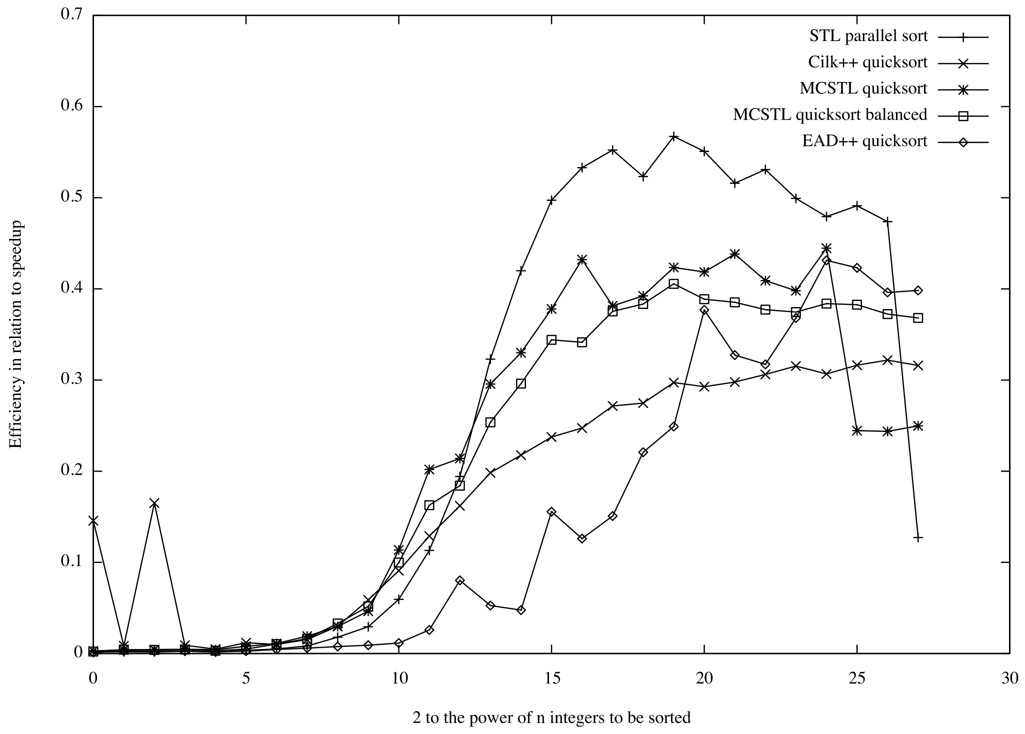
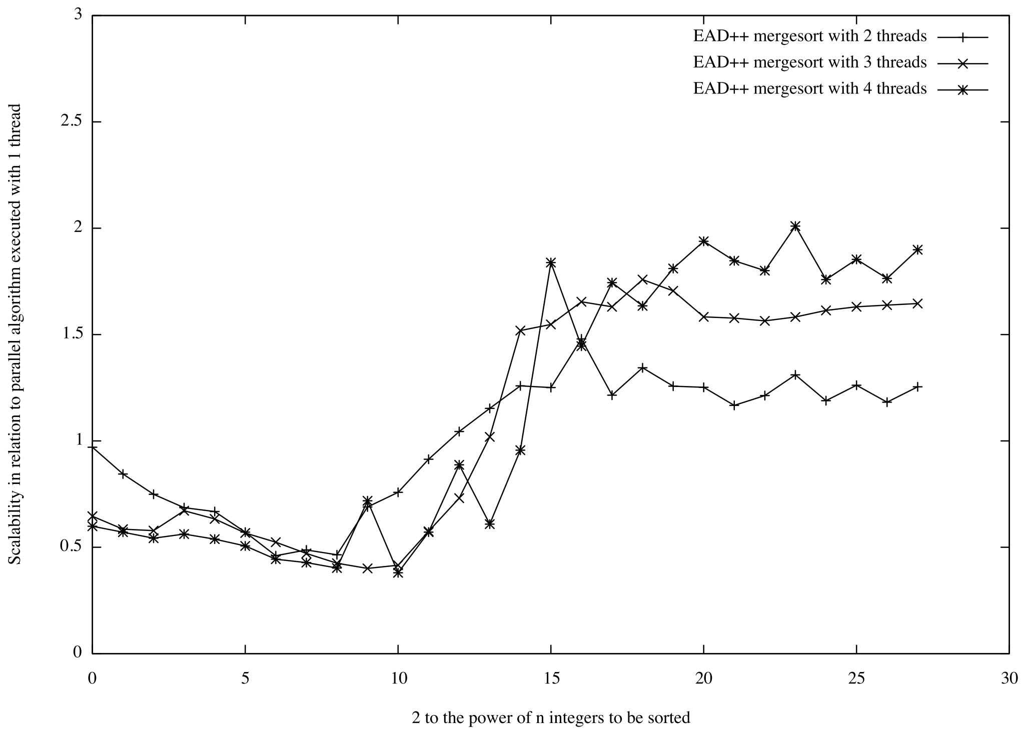


Figure C.4: Quicksort efficiency on dual-hyperthreading with 2 and 4 threads.



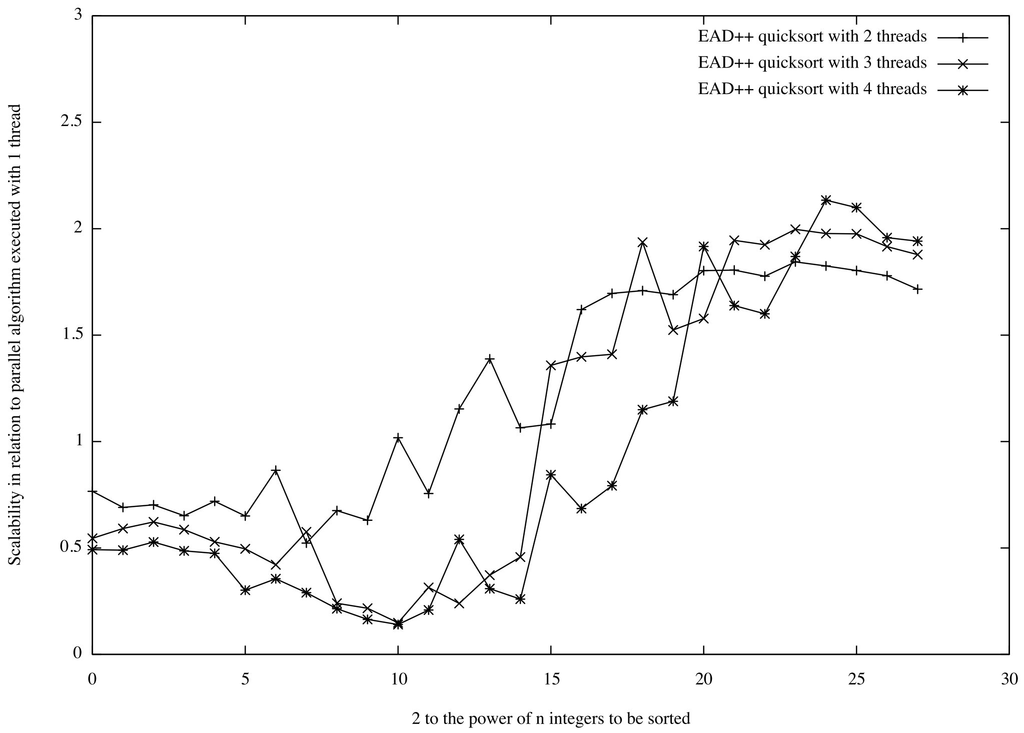
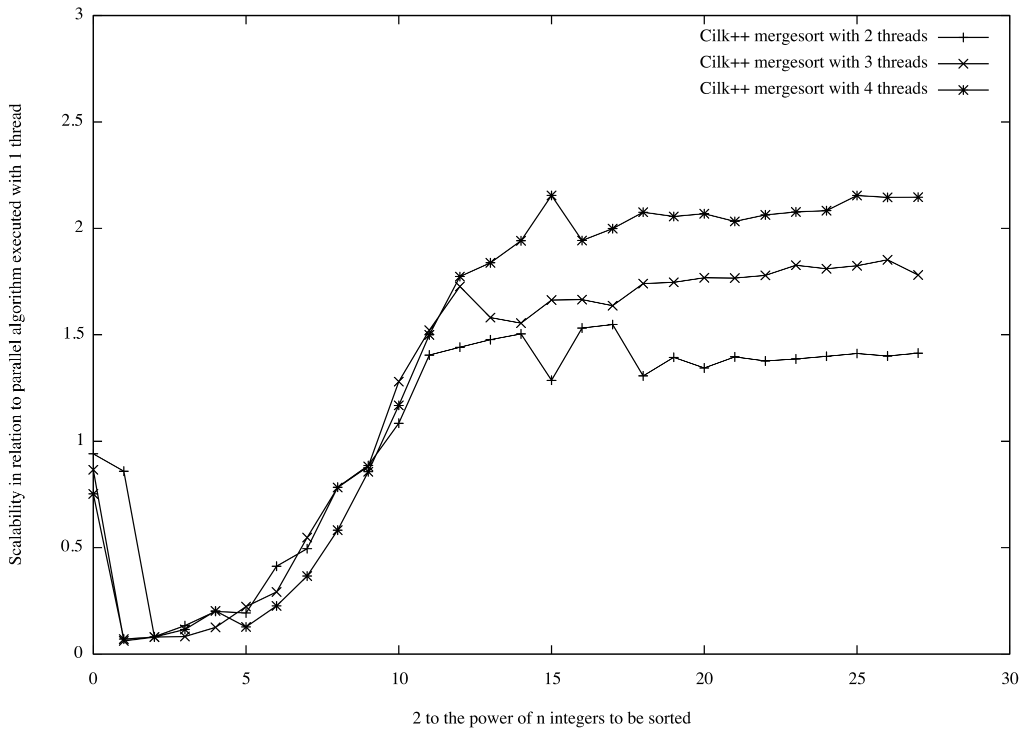


Figure C.5: Scalability of EAD++ mergesort and quicksort on dual-hyperthreading.



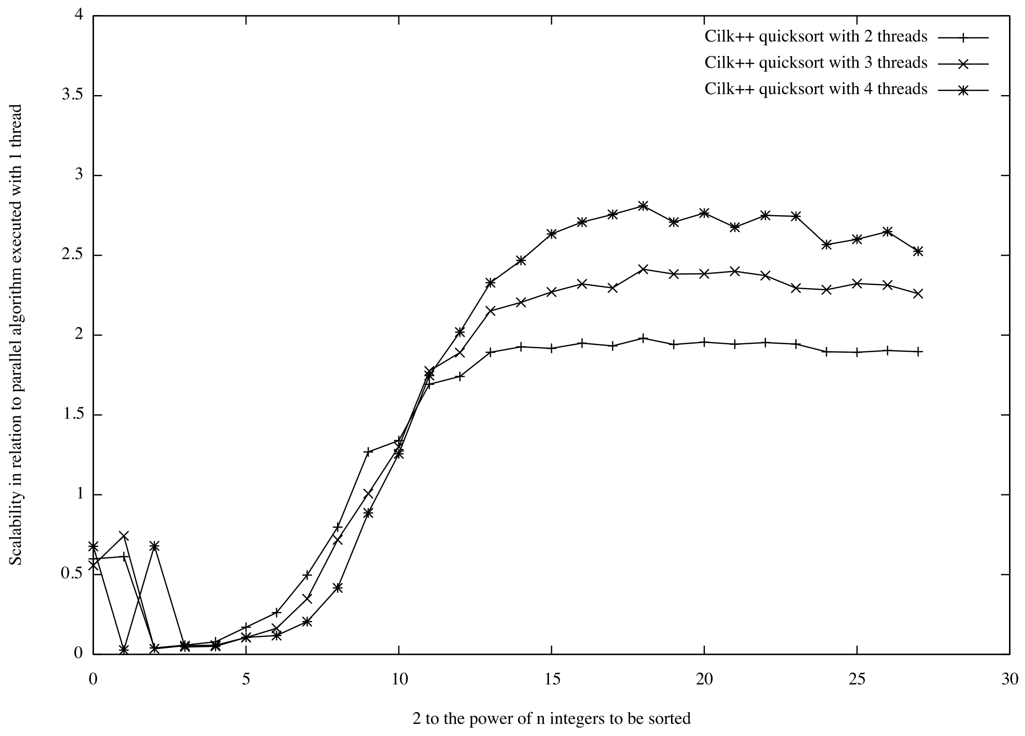
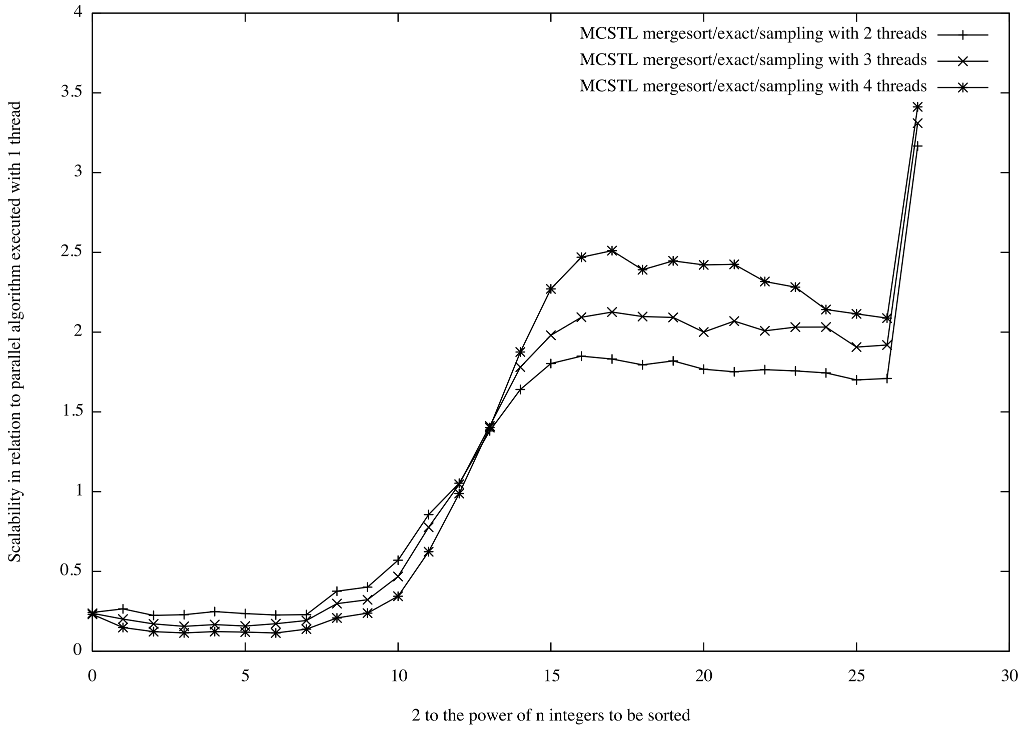


Figure C.6: Scalability of Cilk++ mergesort and quicksort on dual-hyperthreading.



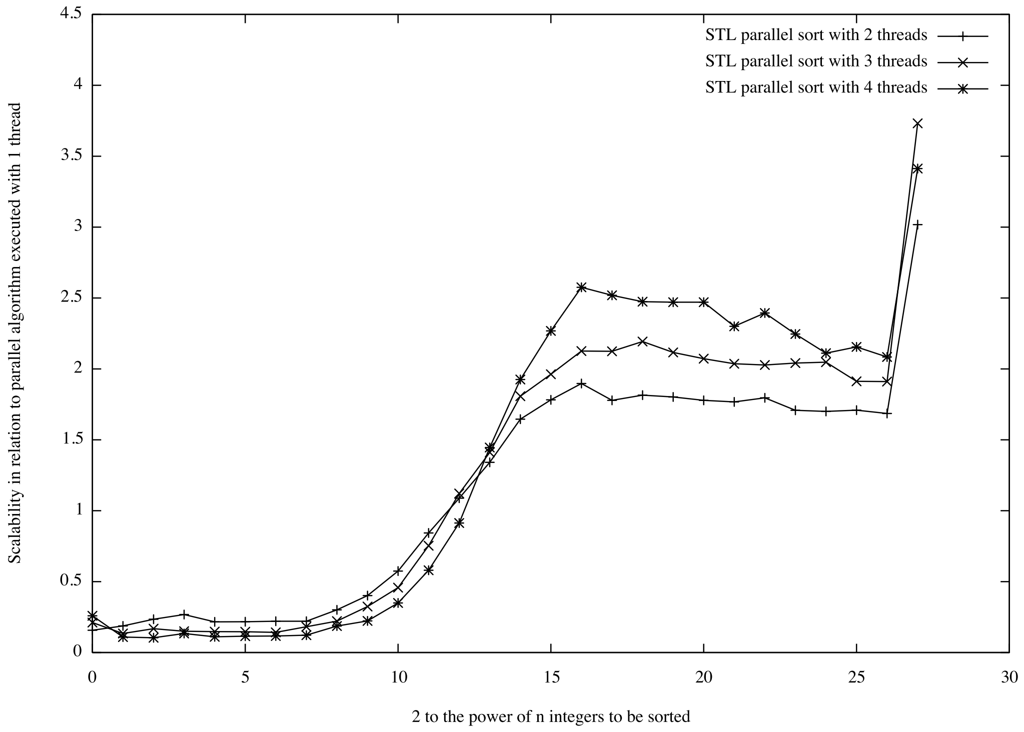
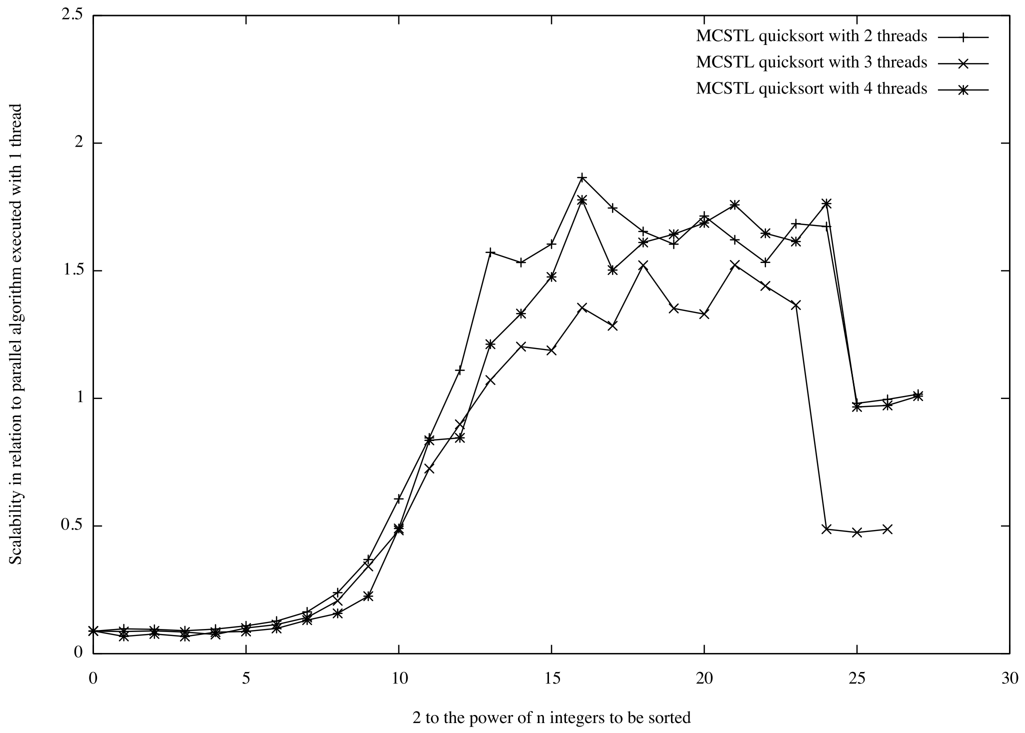


Figure C.7: Scalability of MCSTL mergesort and sort on dual-hyperthreading.



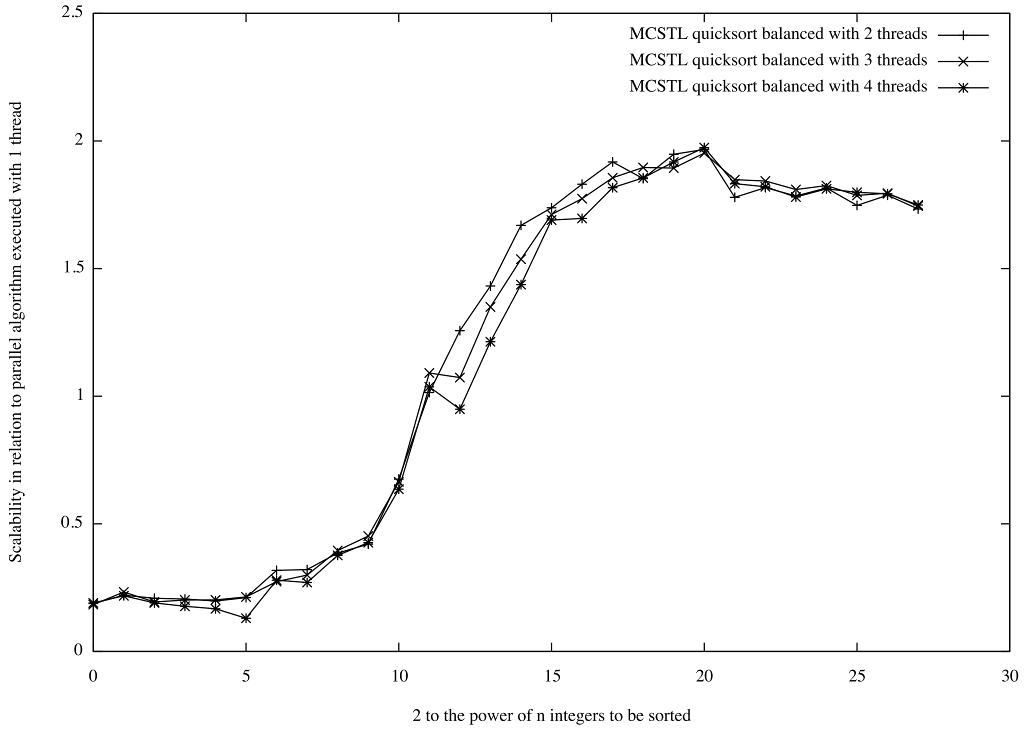
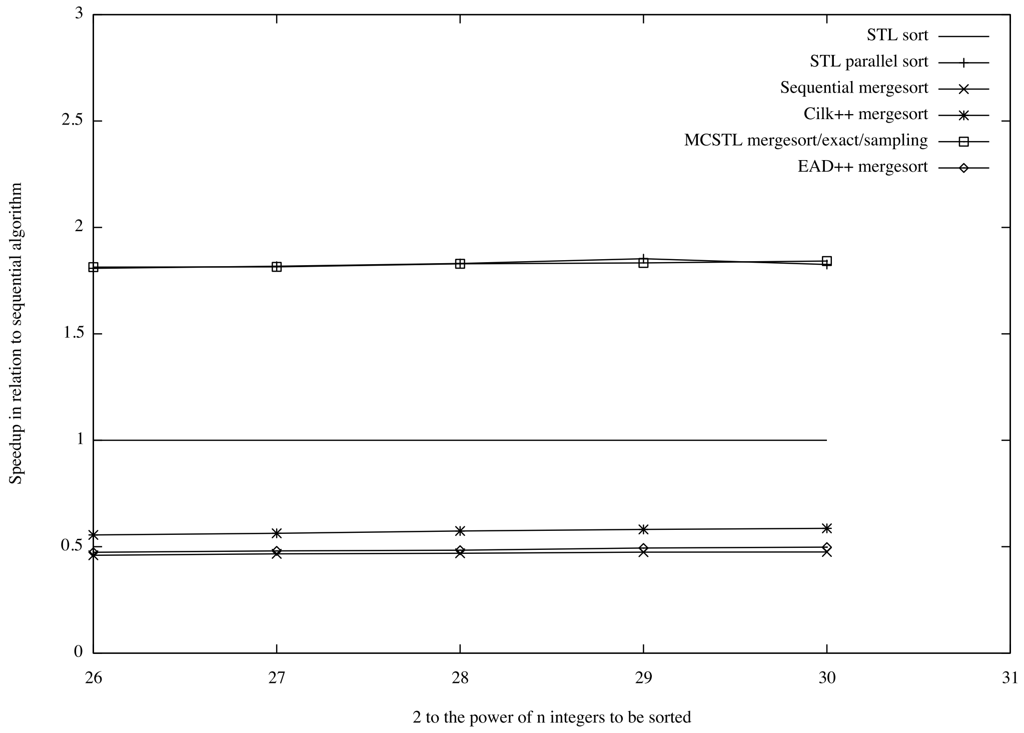


Figure C.8: Scalability of MCSTL quicksort and balanced quicksort on dual-hyperthreading.

C.2 On dual-quadcore server



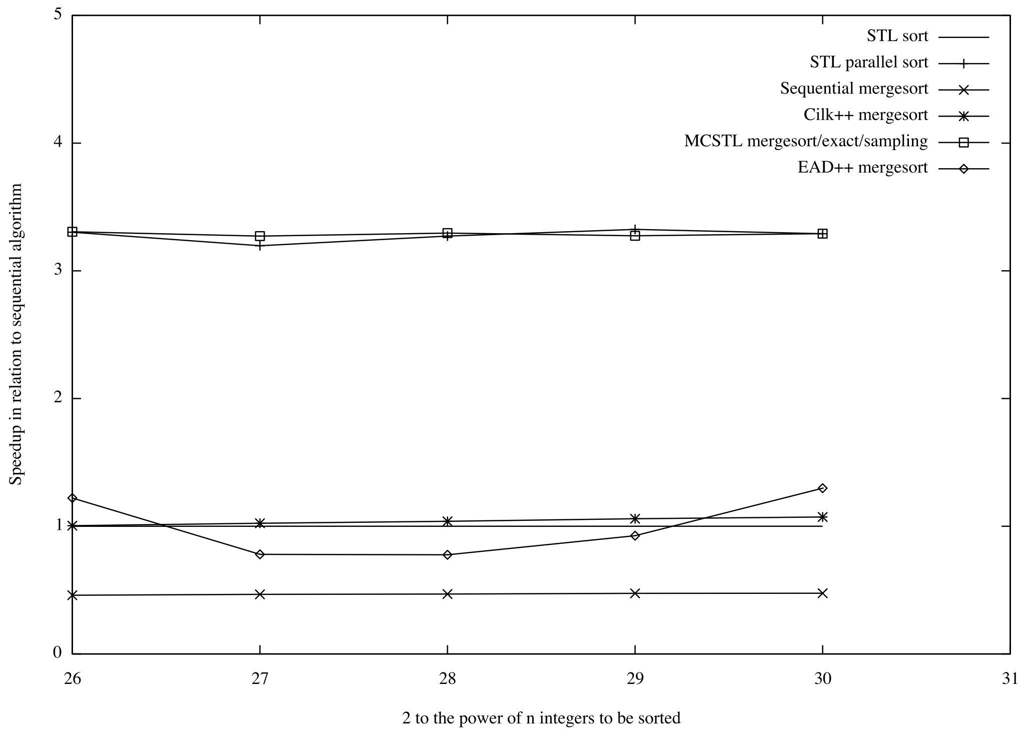


Figure C.9: Mergesort speedup on dual-quadcore with 2 and 4 threads.

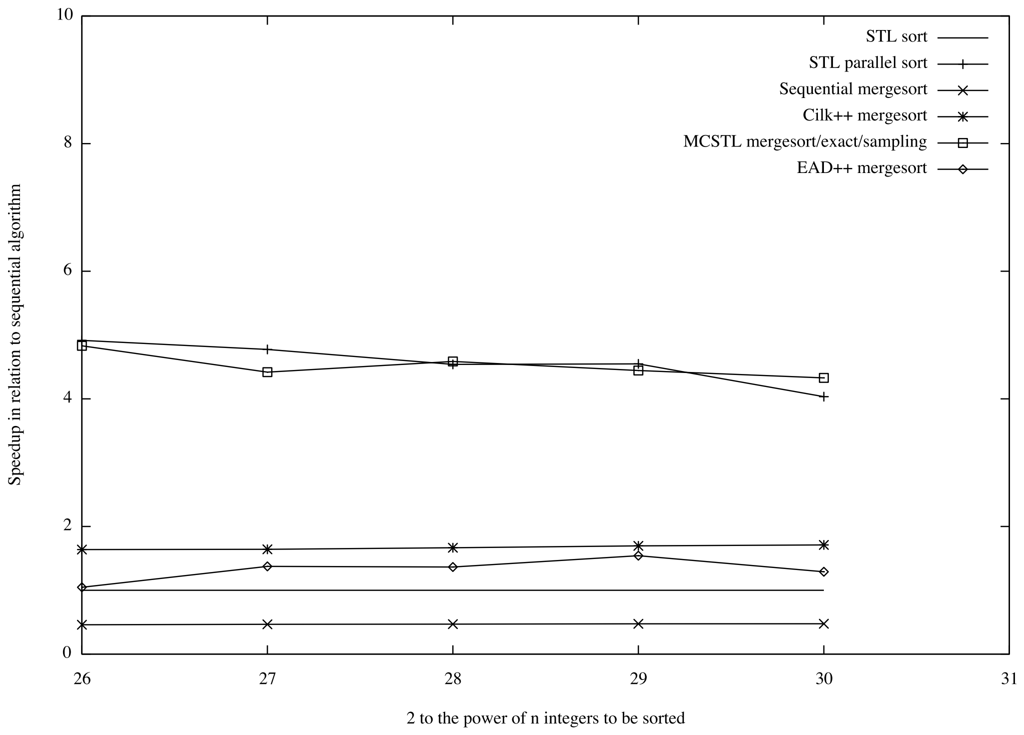
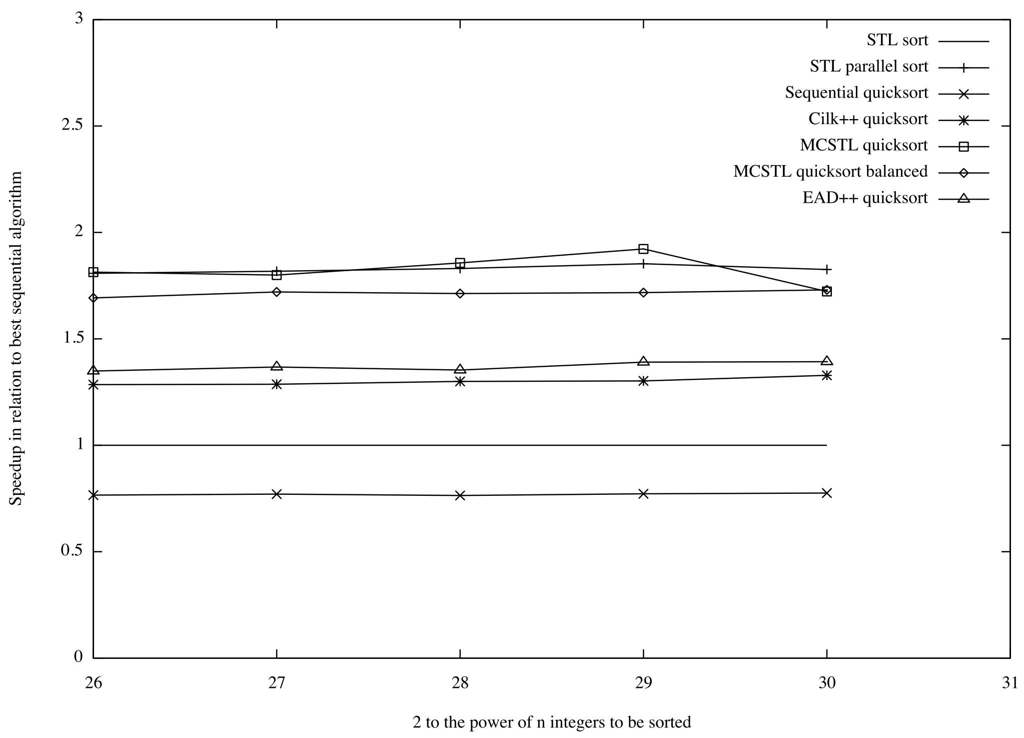


Figure C.10: Mergesort speedup on dual-quadcore with 8 threads.



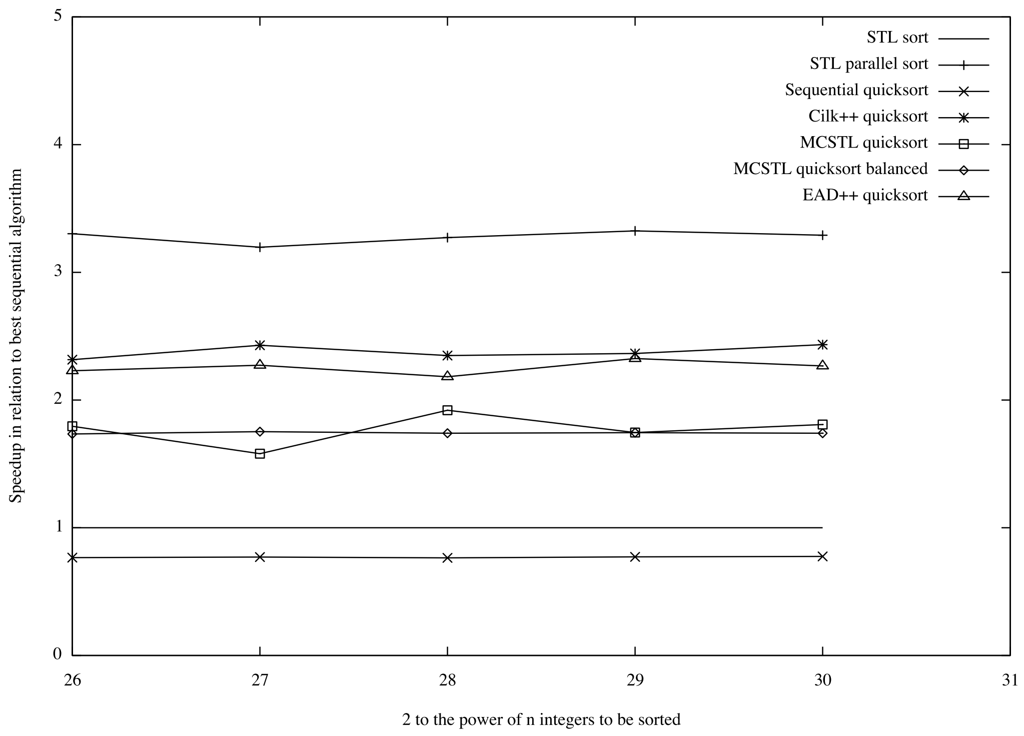


Figure C.11: Quicksort speedup on dual-quadcore with 2 and 4 threads.

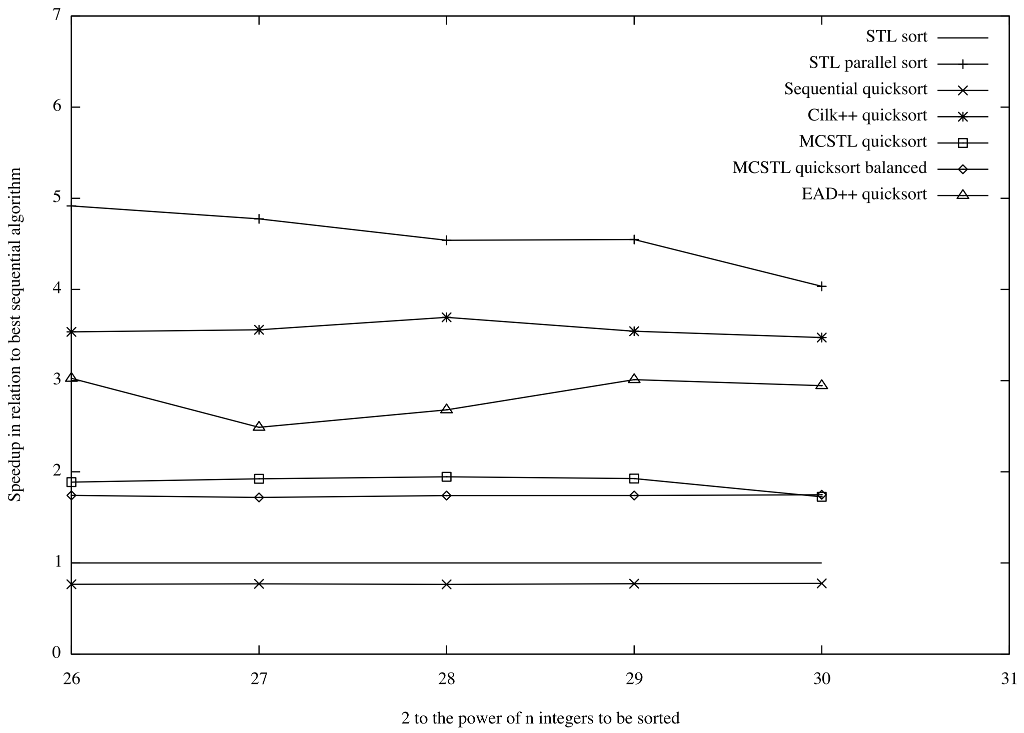
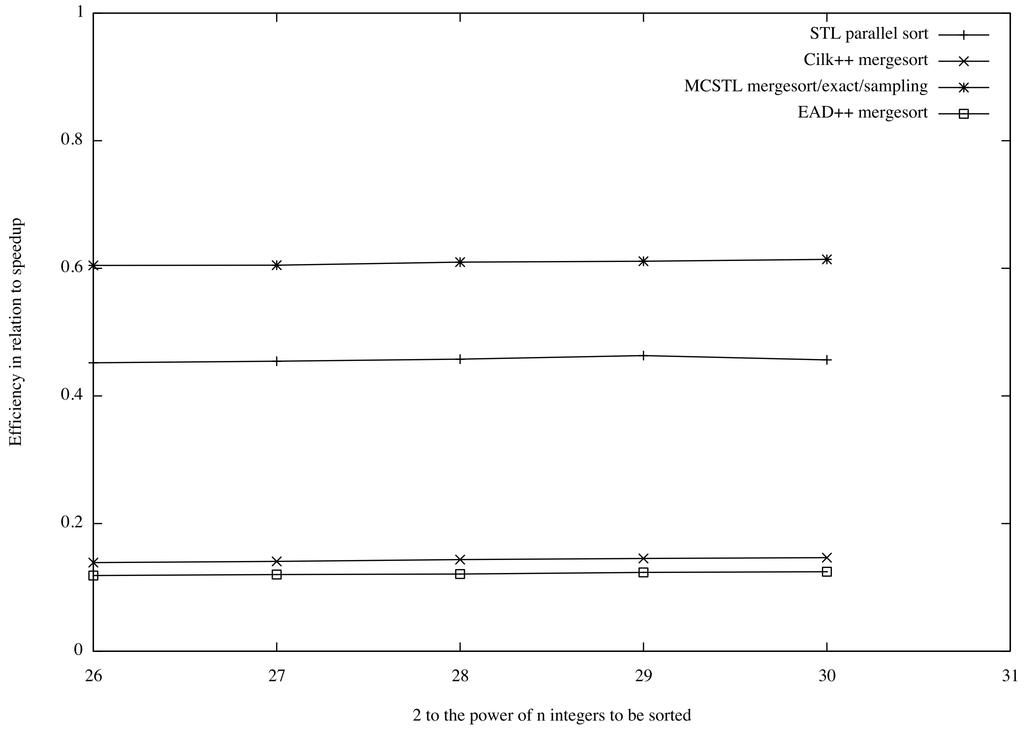


Figure C.12: Quicksort speedup on dual-quadcore with 8 threads.



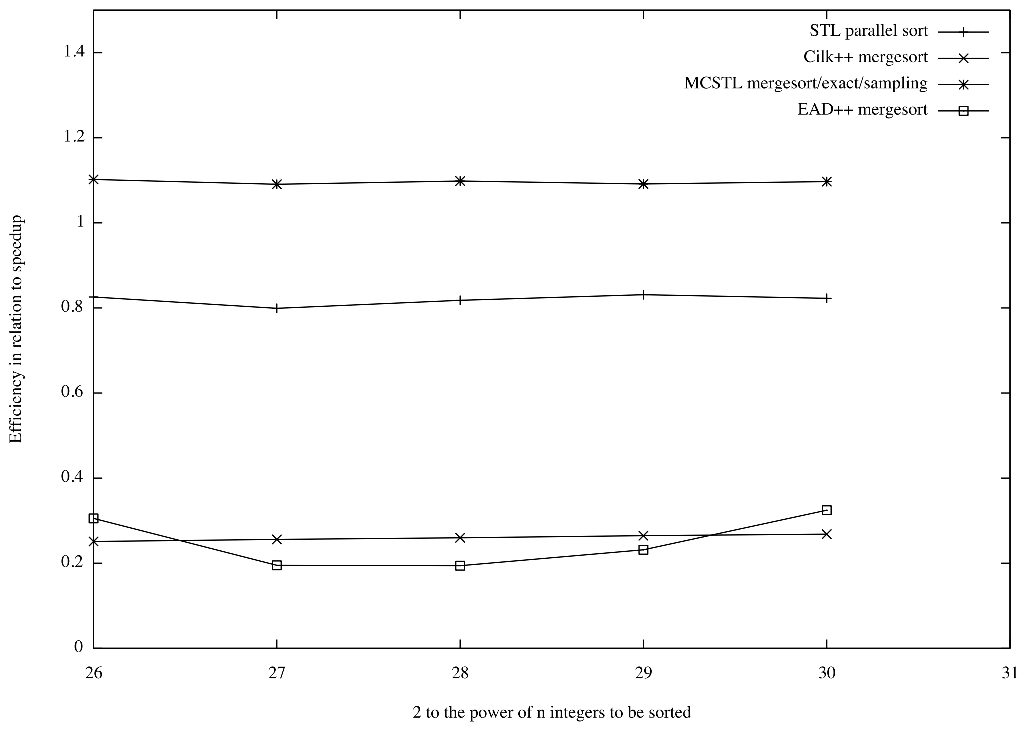


Figure C.13: Mergesort efficiency on dual-quadcore with 2 and 4 threads.

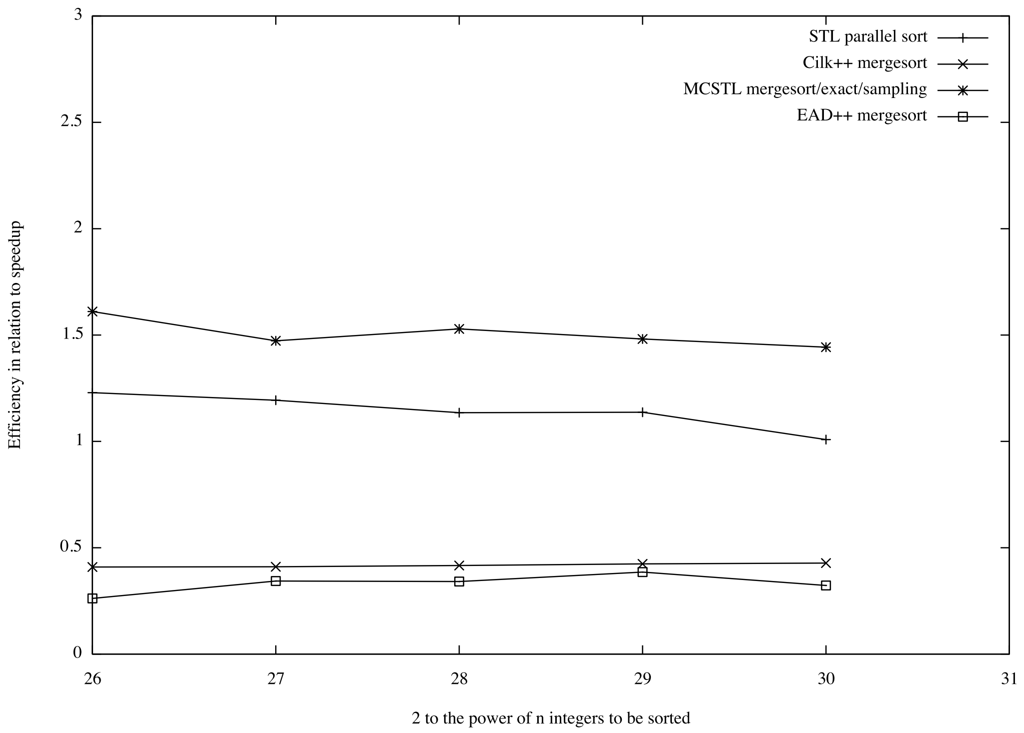
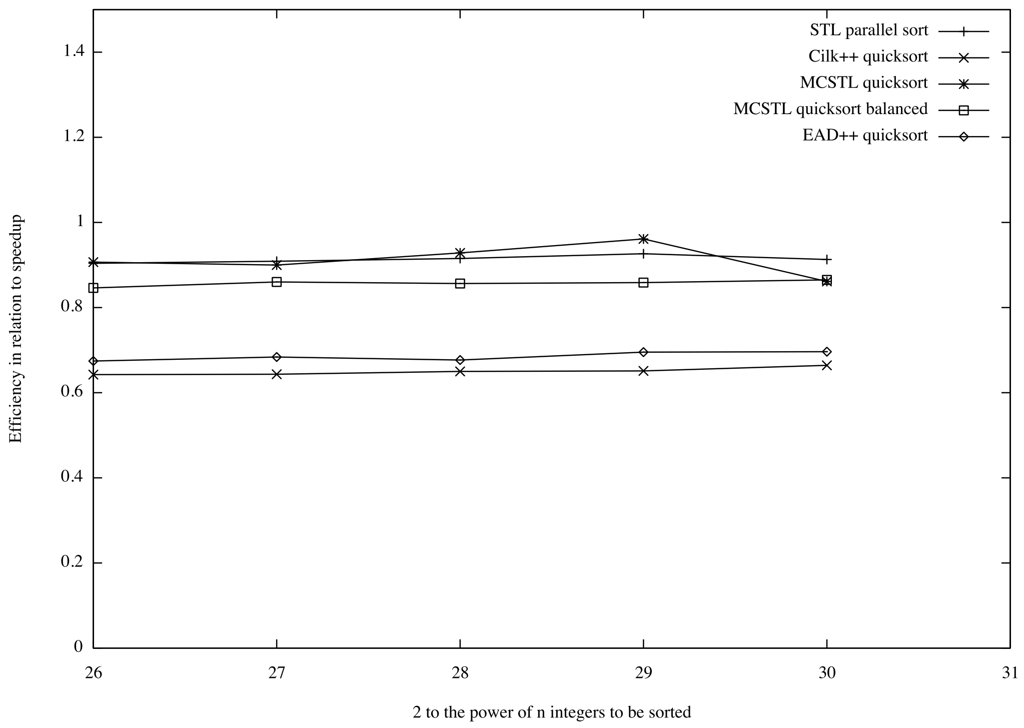


Figure C.14: Mergesort efficiency on dual-quadcore with 8 threads.



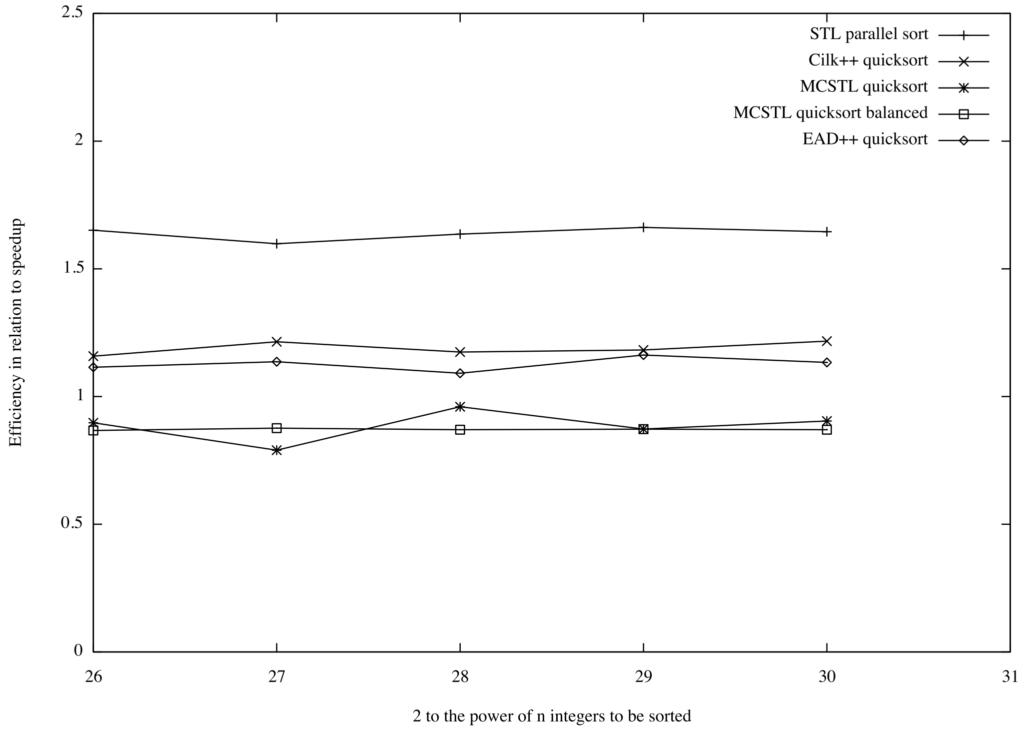


Figure C.15: Quicksort efficiency on dual-quadcore with 2 and 4 threads.

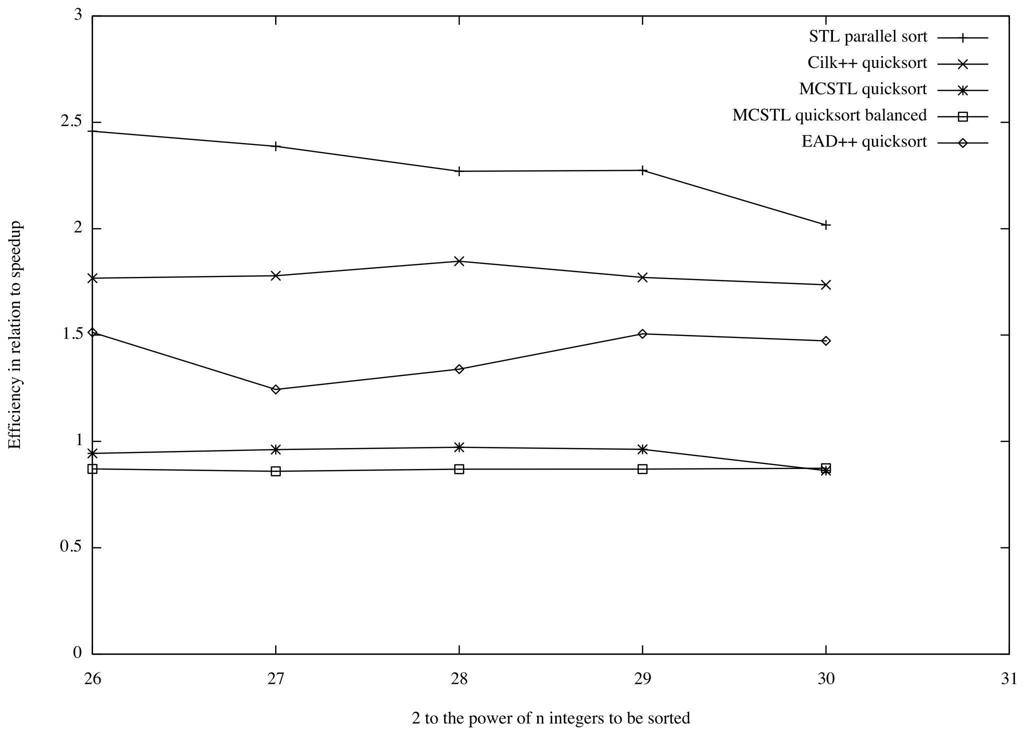
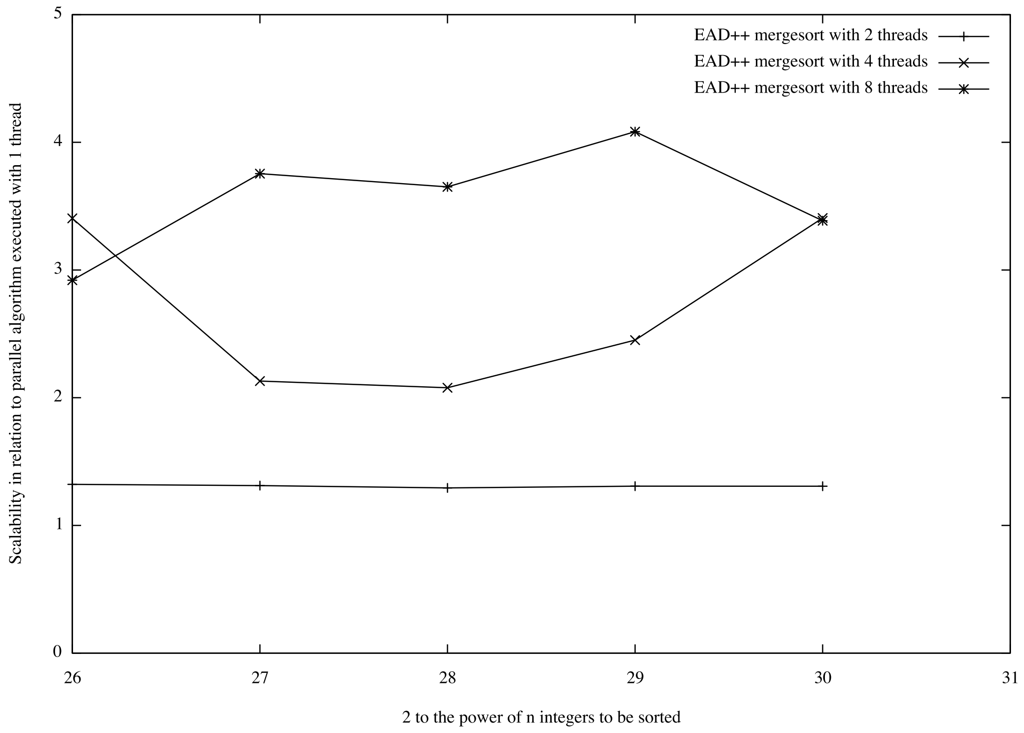


Figure C.16: Quicksort efficiency on dual-quadcore with 8 threads.



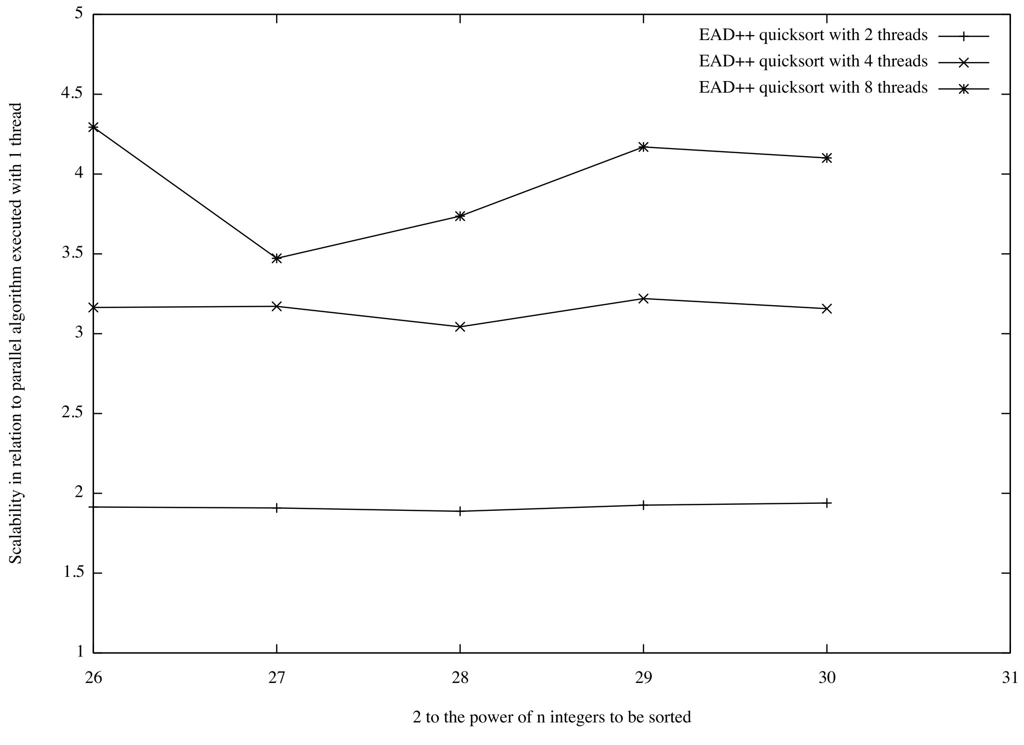
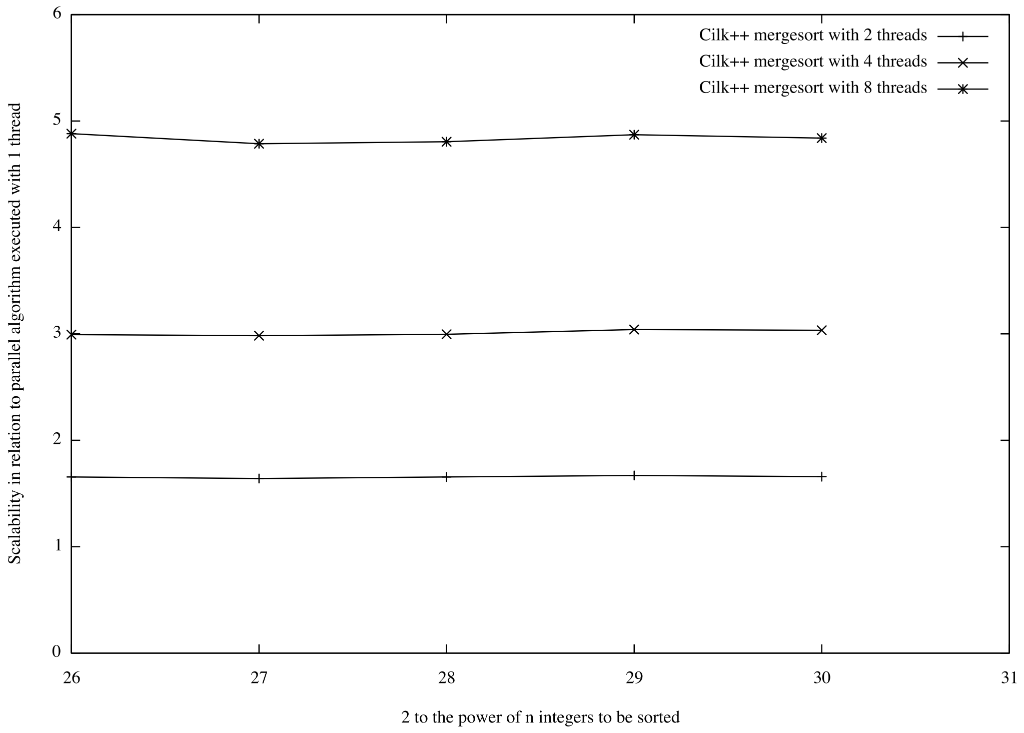


Figure C.17: Scalability of EAD++ mergesort and quicksort on dual-quadcore.



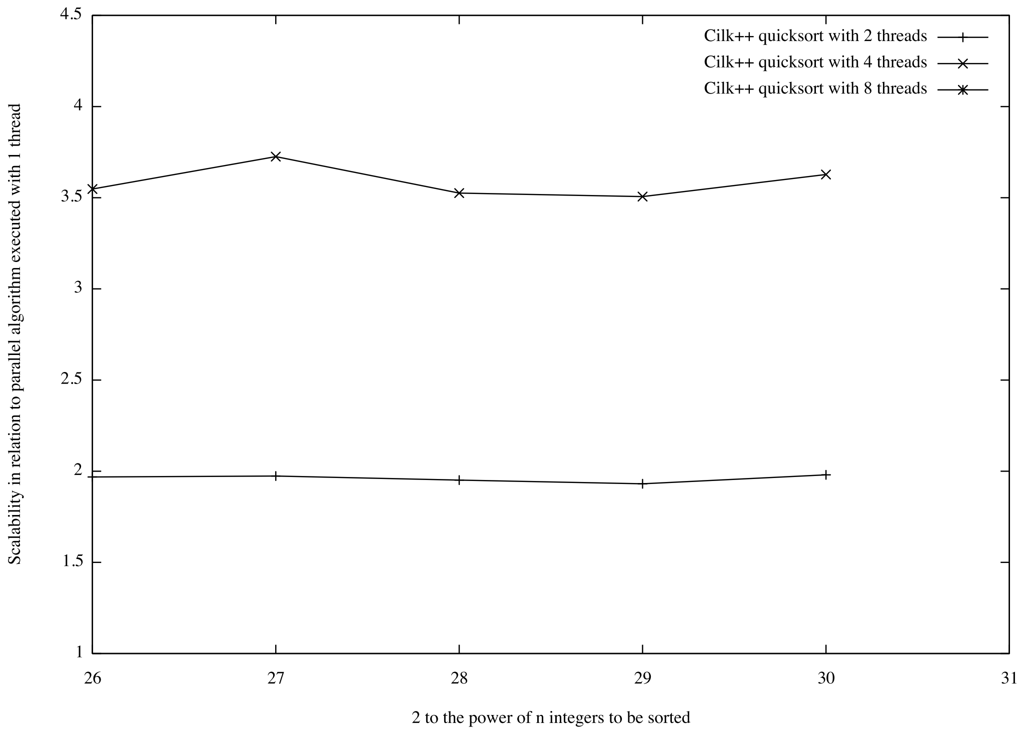
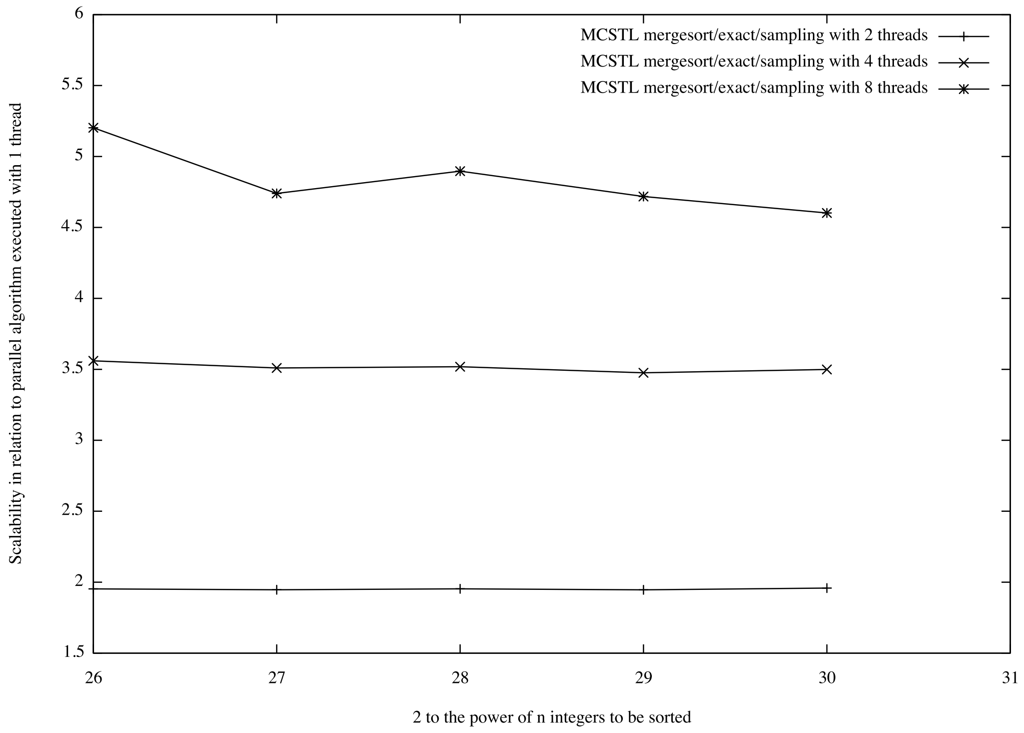


Figure C.18: Scalability of Cilk++ mergesort and quicksort on dual-quadcore.



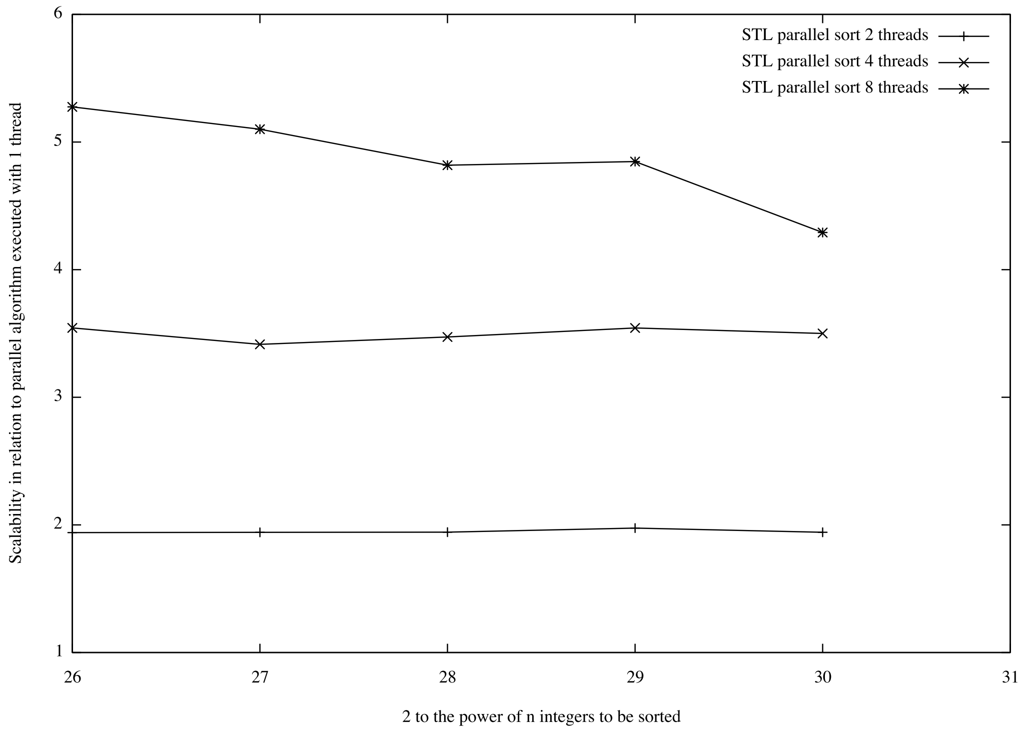
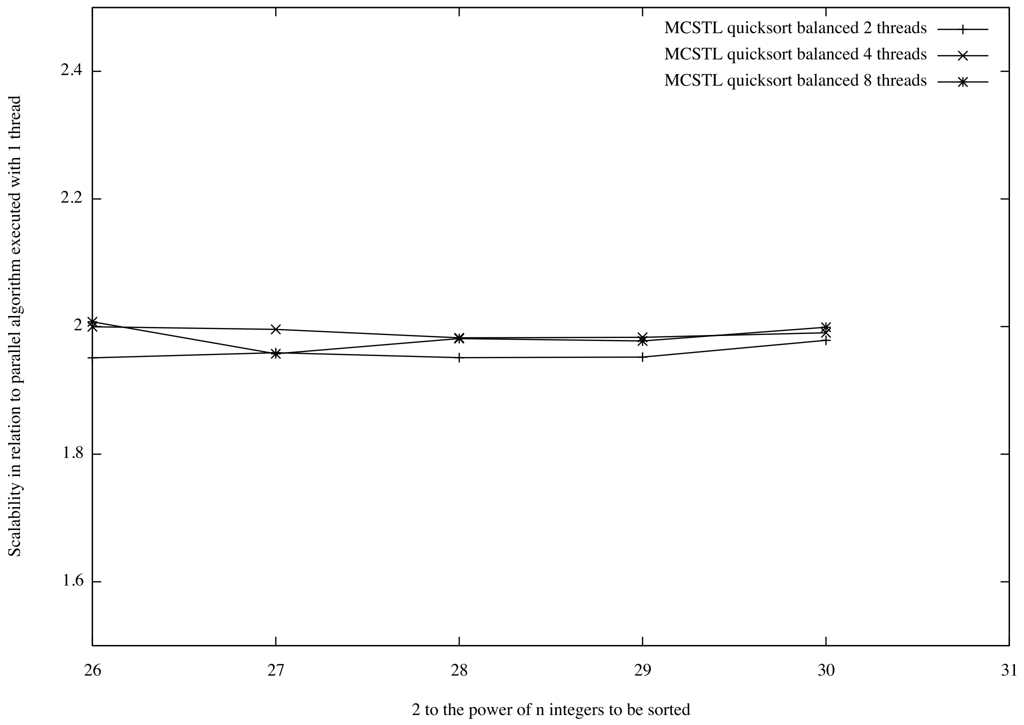


Figure C.19: Scalability of MCSTL mergesort and sort on dual-quadcore.



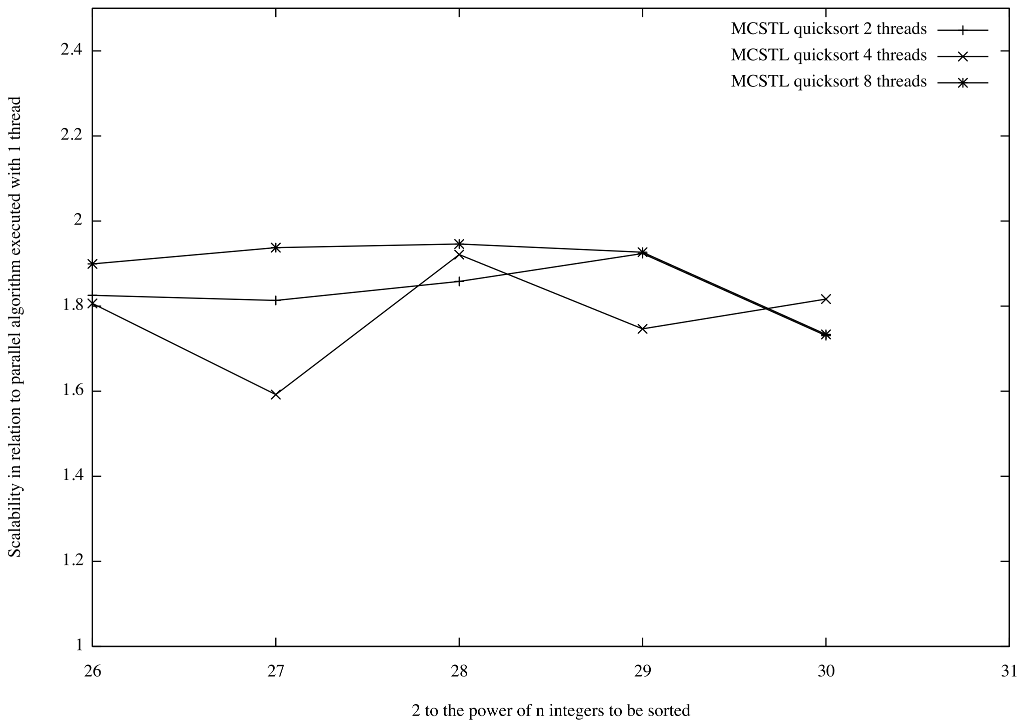
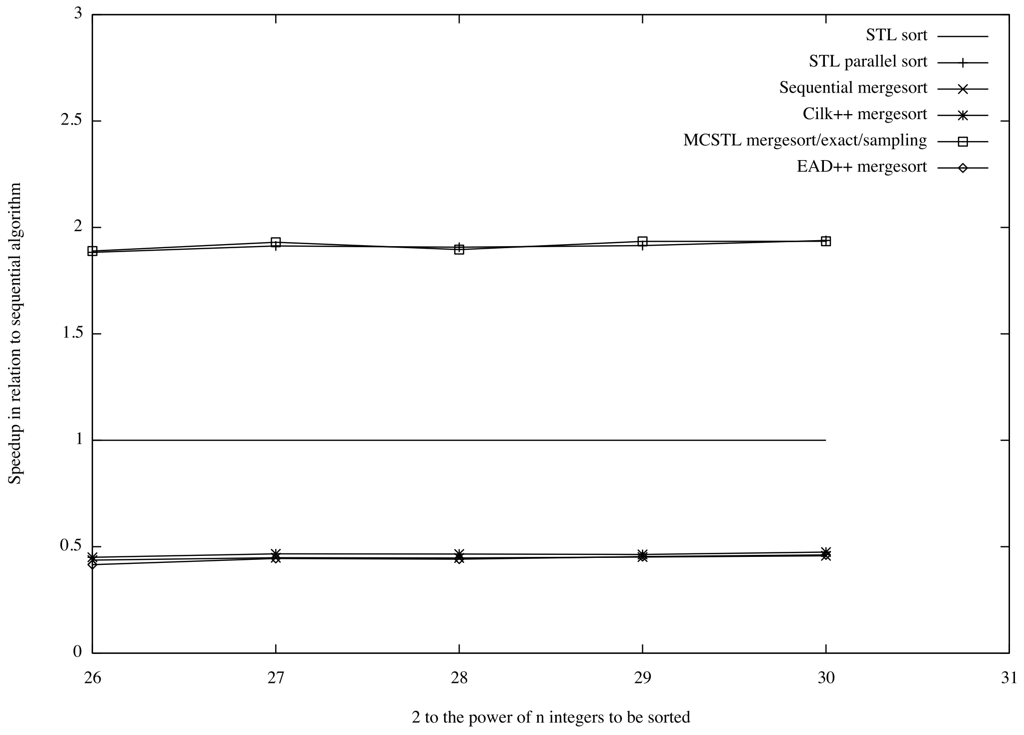


Figure C.20: Scalability of MCSTL quicksort and balanced quicksort on dual-quadcore.

C.3 On tetra-octacore server



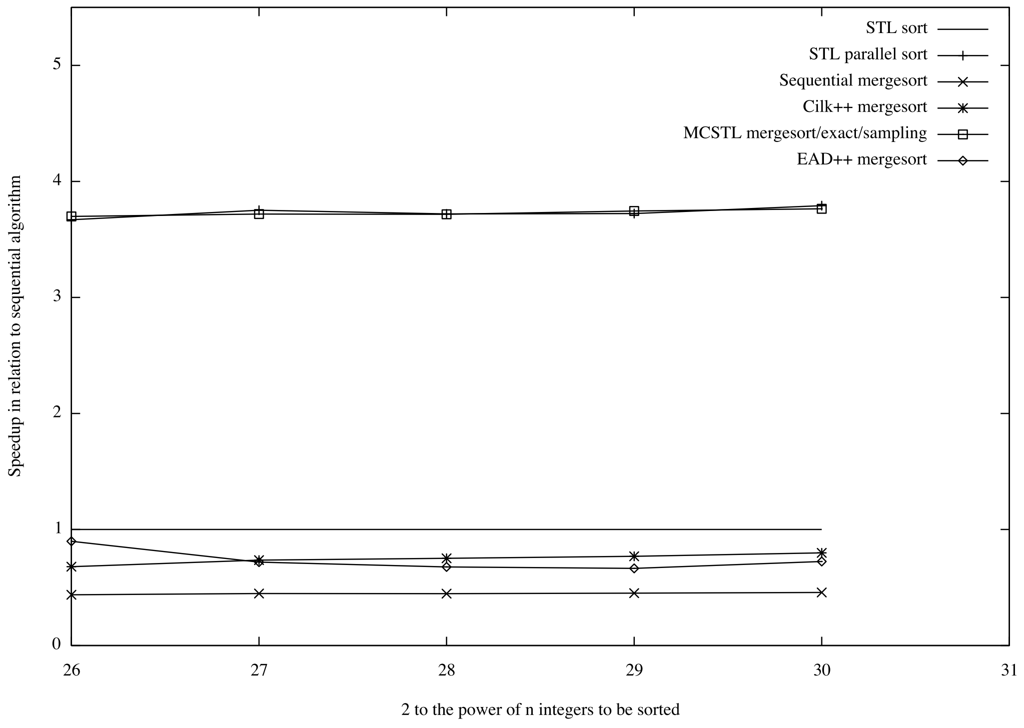
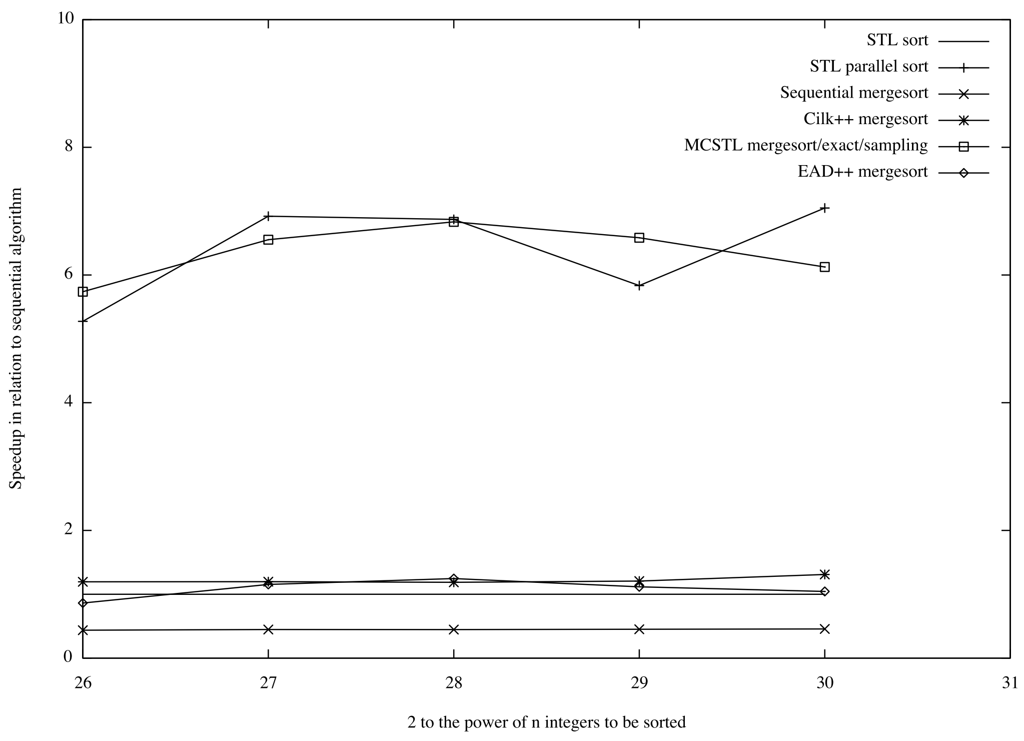


Figure C.21: Mergesort speedup on tetra-octacore with 2 and 4 threads.



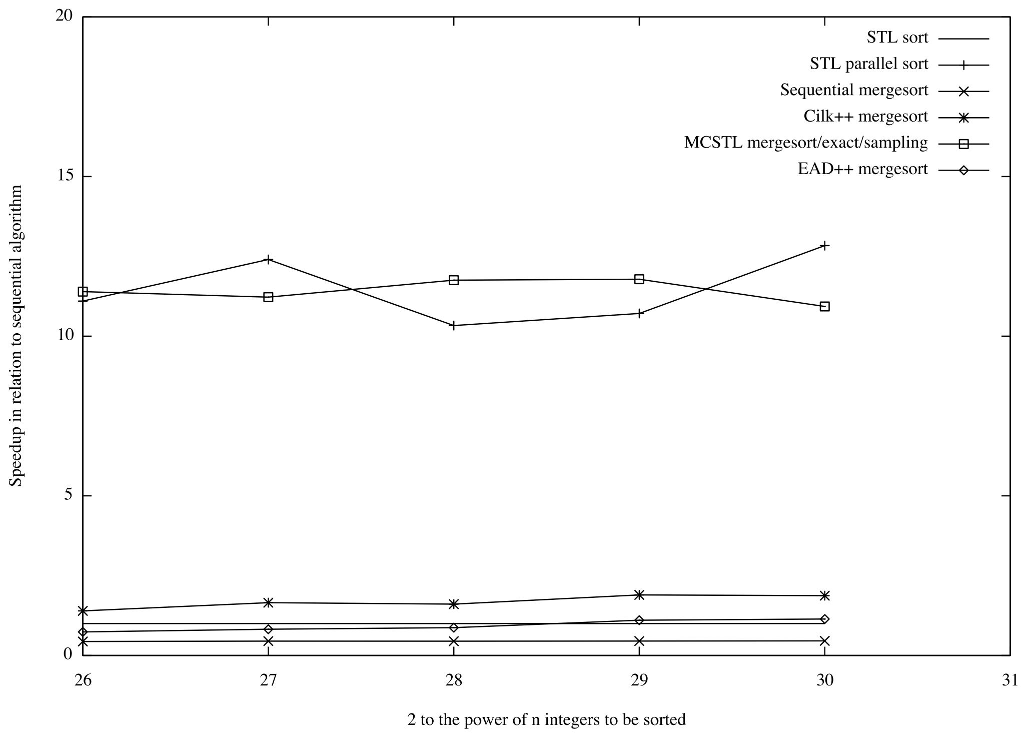


Figure C.22: Mergesort speedup on tetra-octacore with 8 and 16 threads.

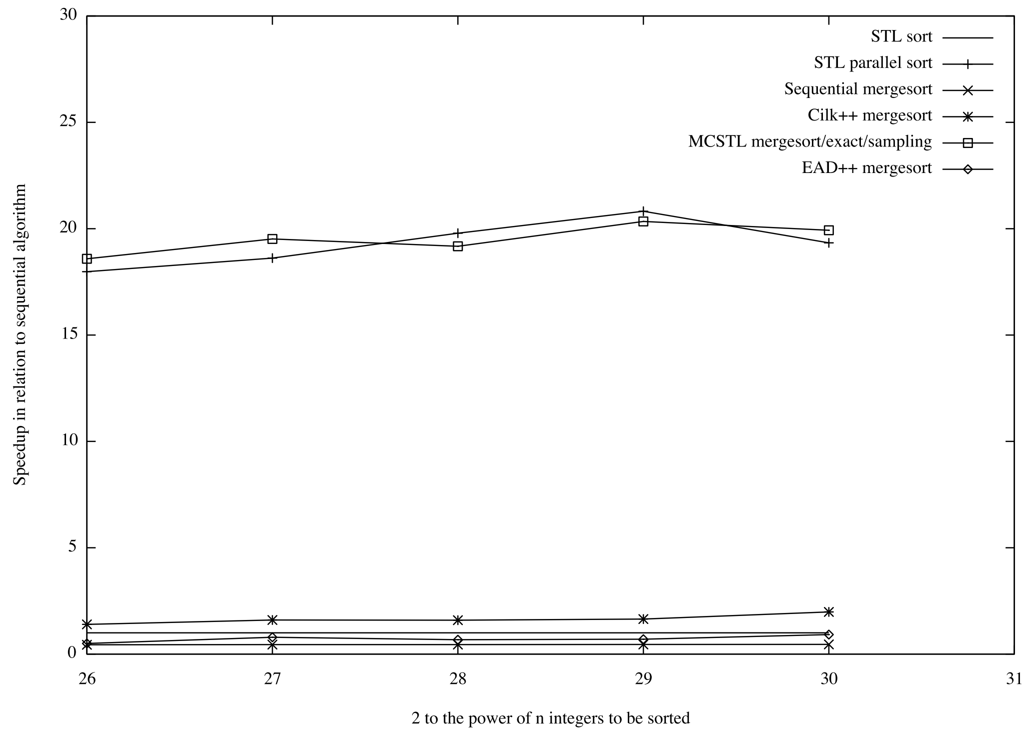
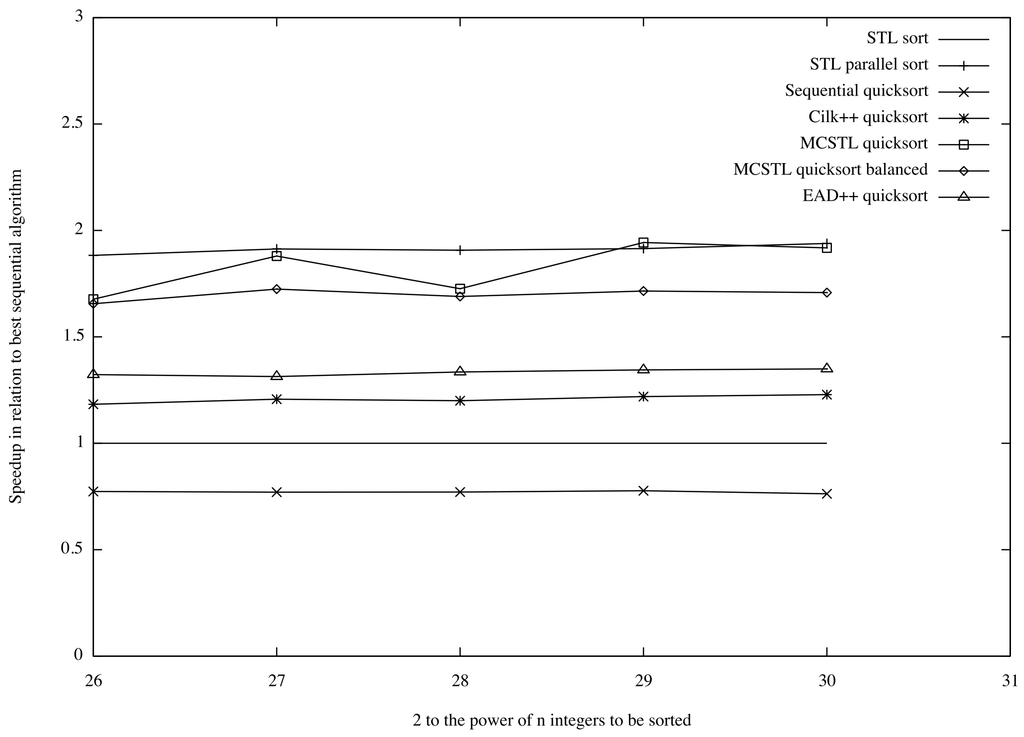


Figure C.23: Mergesort speedup on tetra-octacore with 32 threads.



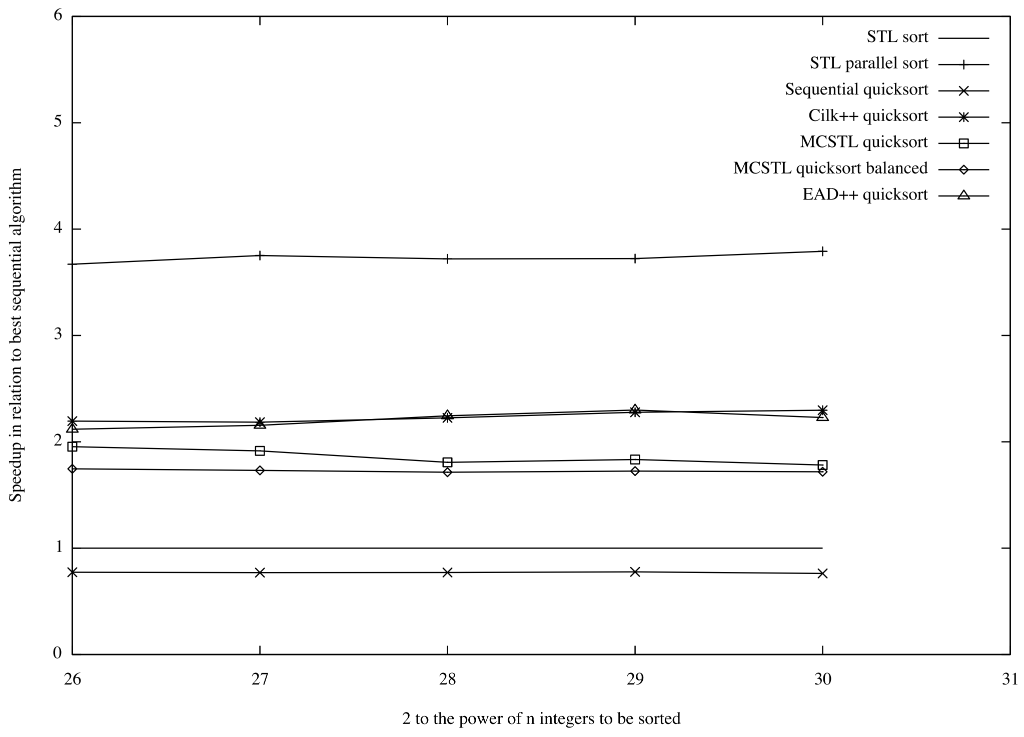
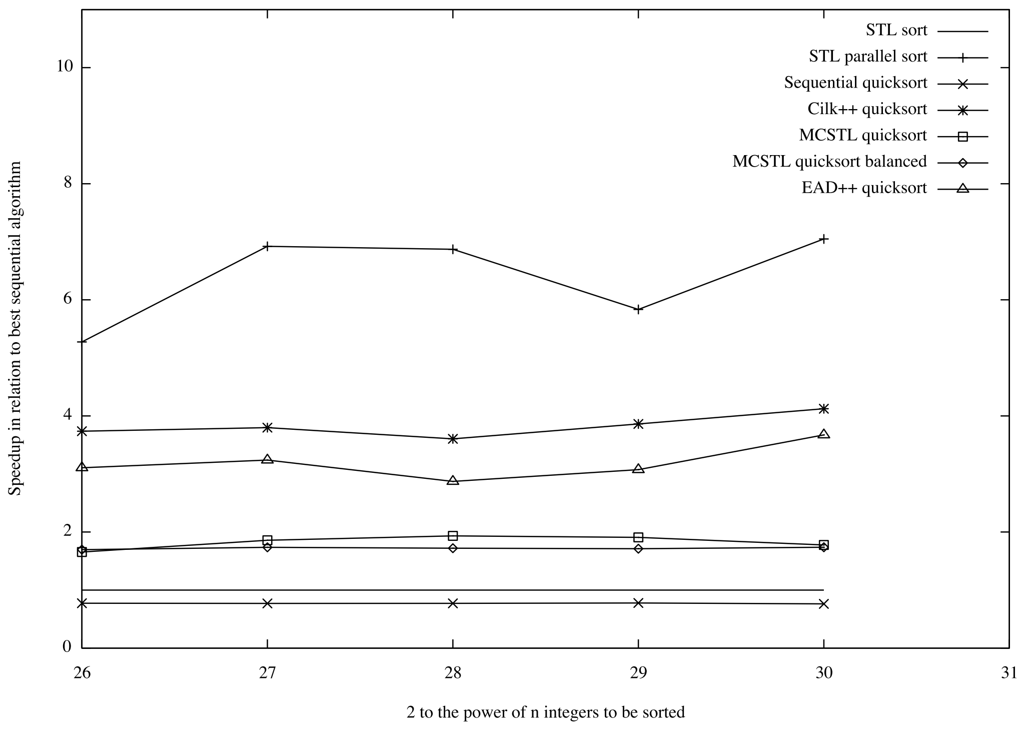


Figure C.24: Quicksort speedup on tetra-octacore with 2 and 4 threads.



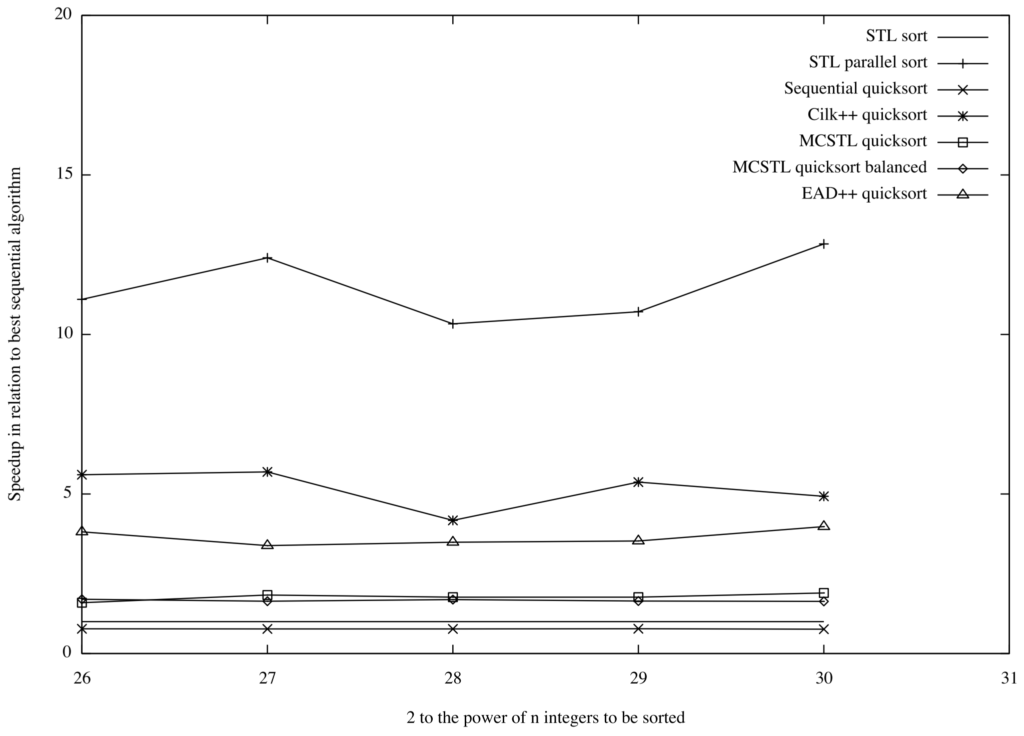


Figure C.25: Quicksort speedup on tetra-octacore with 8 and 16 threads.

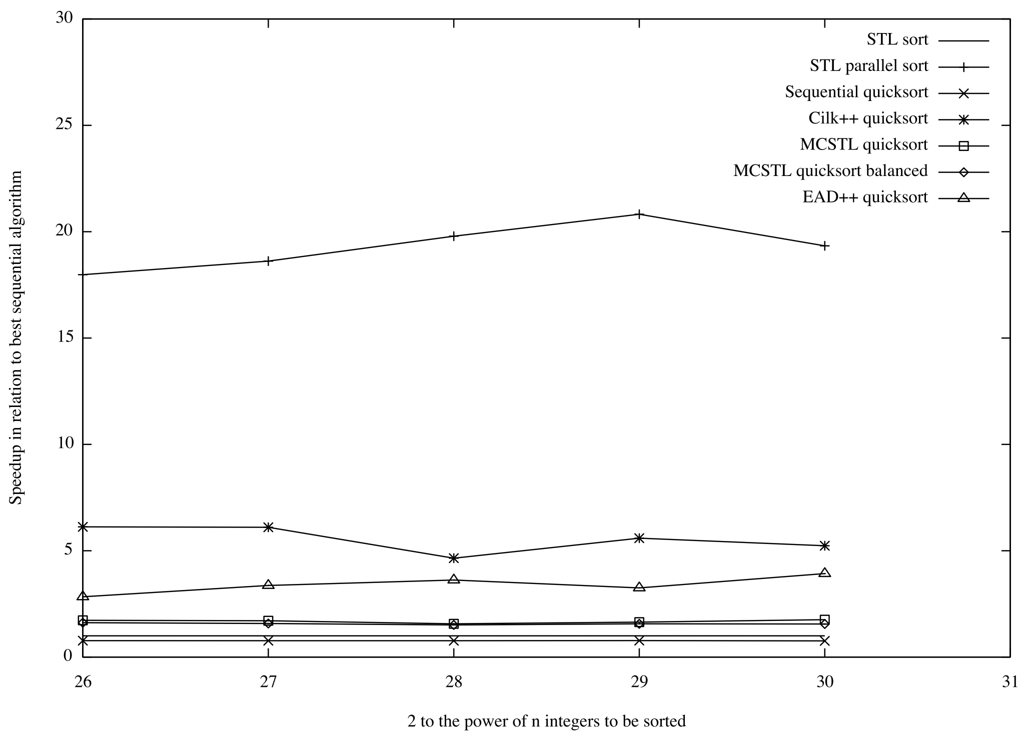
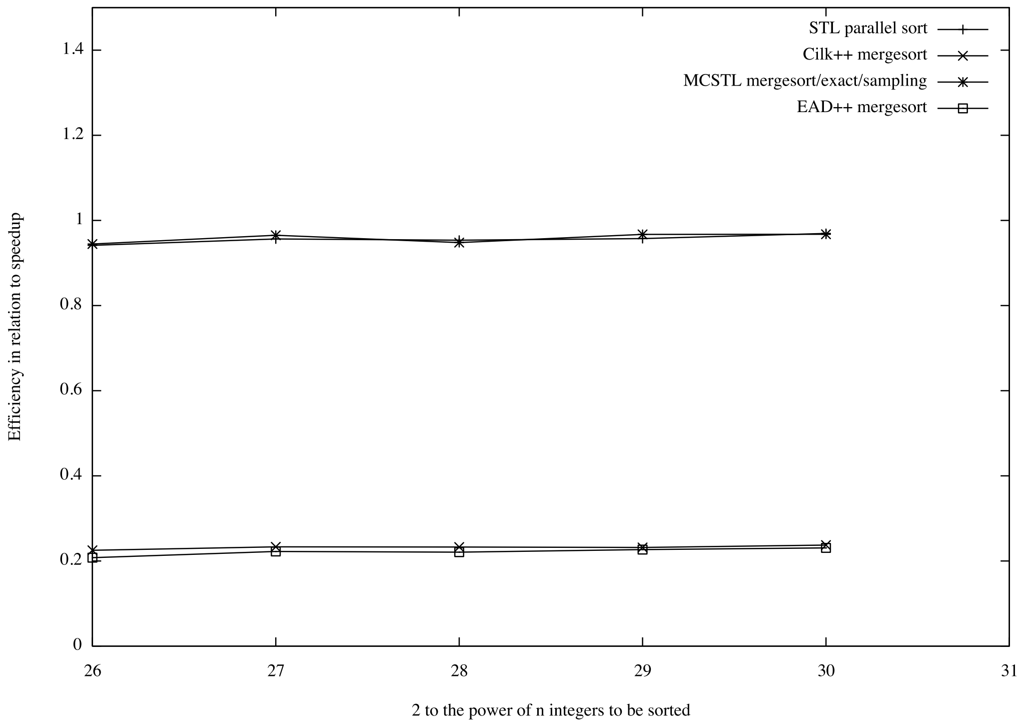


Figure C.26: Quicksort speedup on tetra-octacore with 32 threads.



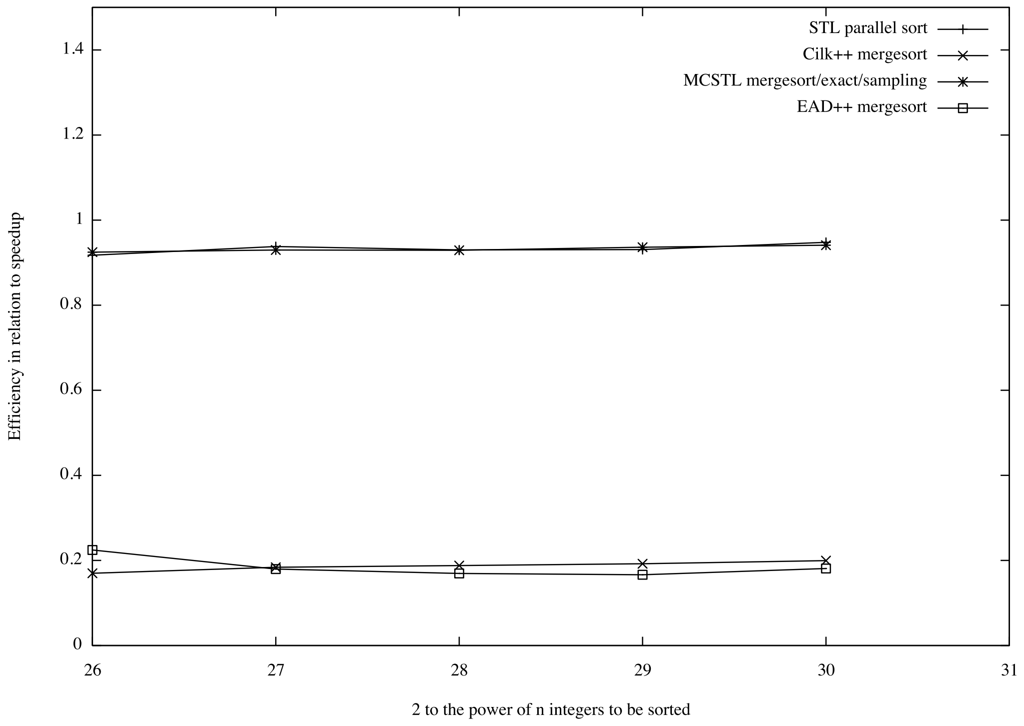
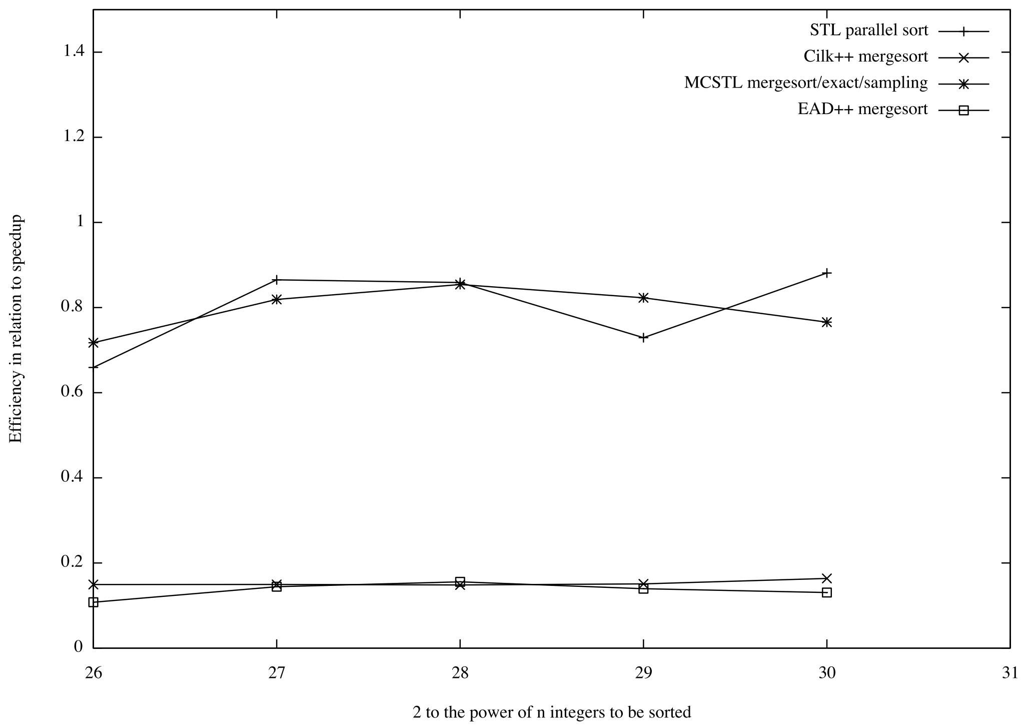


Figure C.27: Mergesort efficiency on tetra-octacore with 2 and 4 threads.



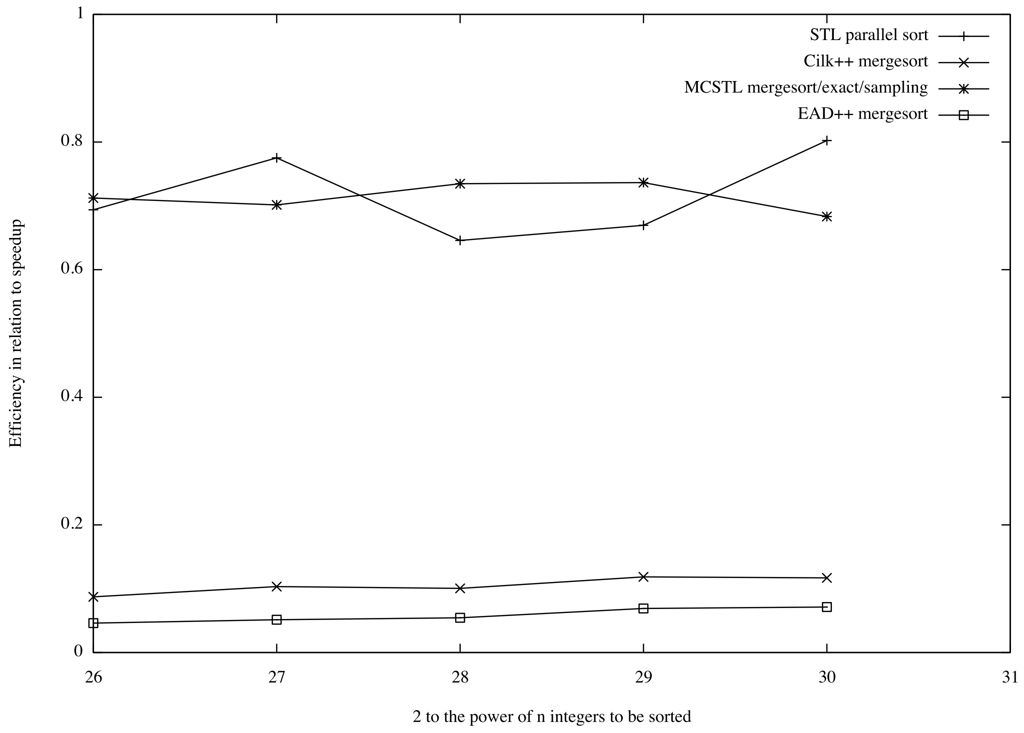


Figure C.28: Mergesort efficiency on tetra-octacore with 8 and 16 threads.

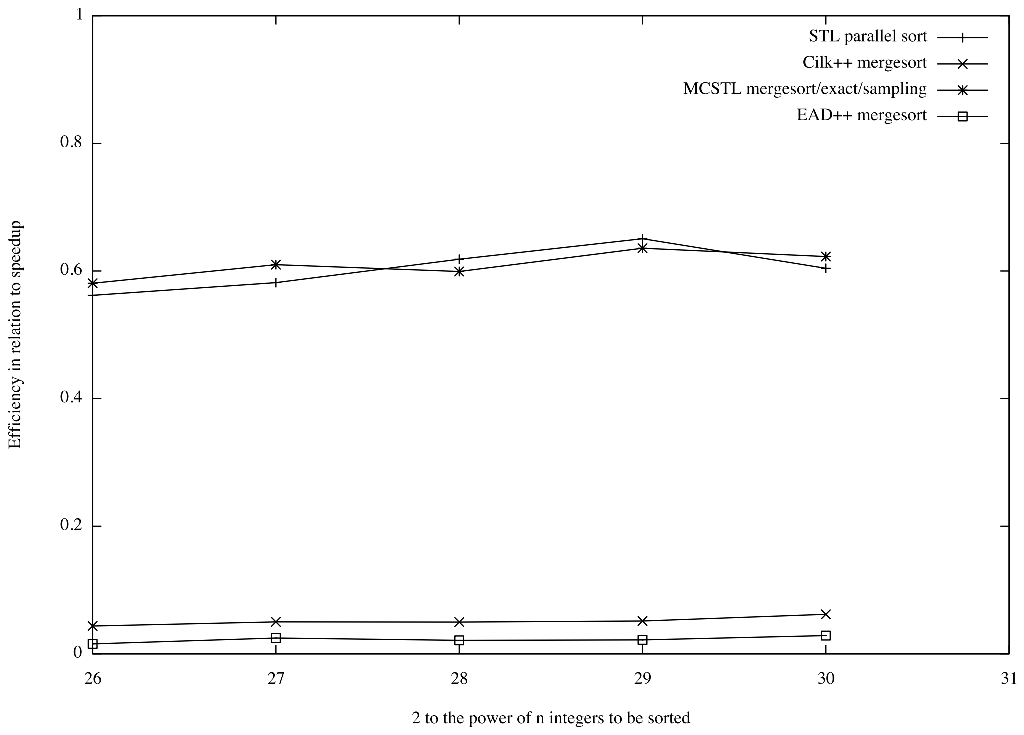
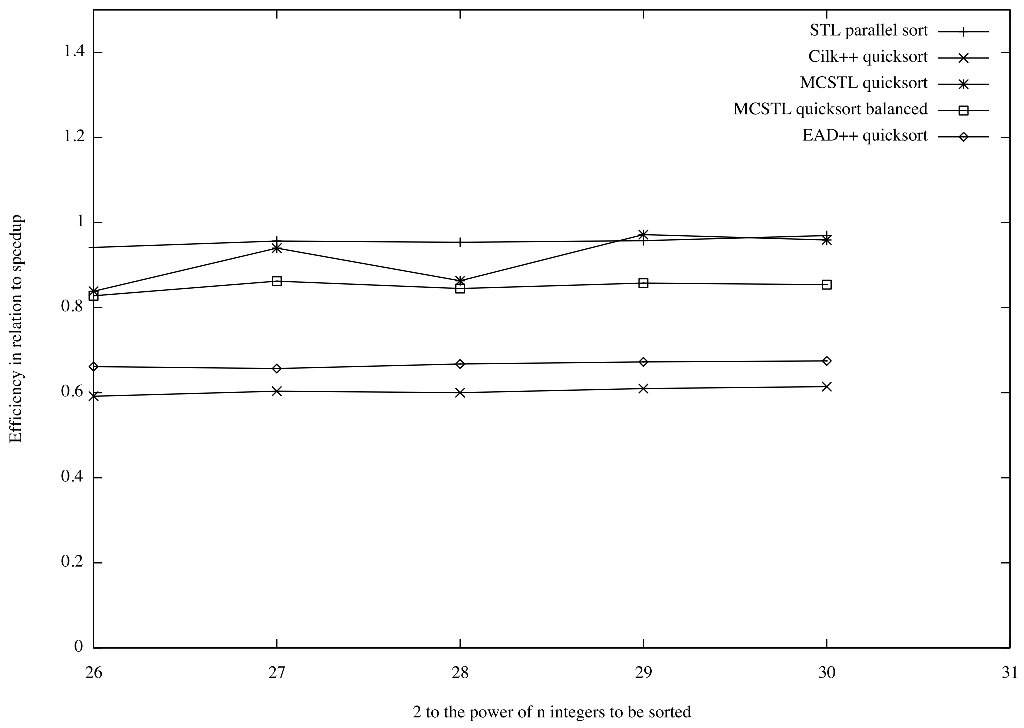


Figure C.29: Mergesort efficiency on tetra-octacore with 32 threads.



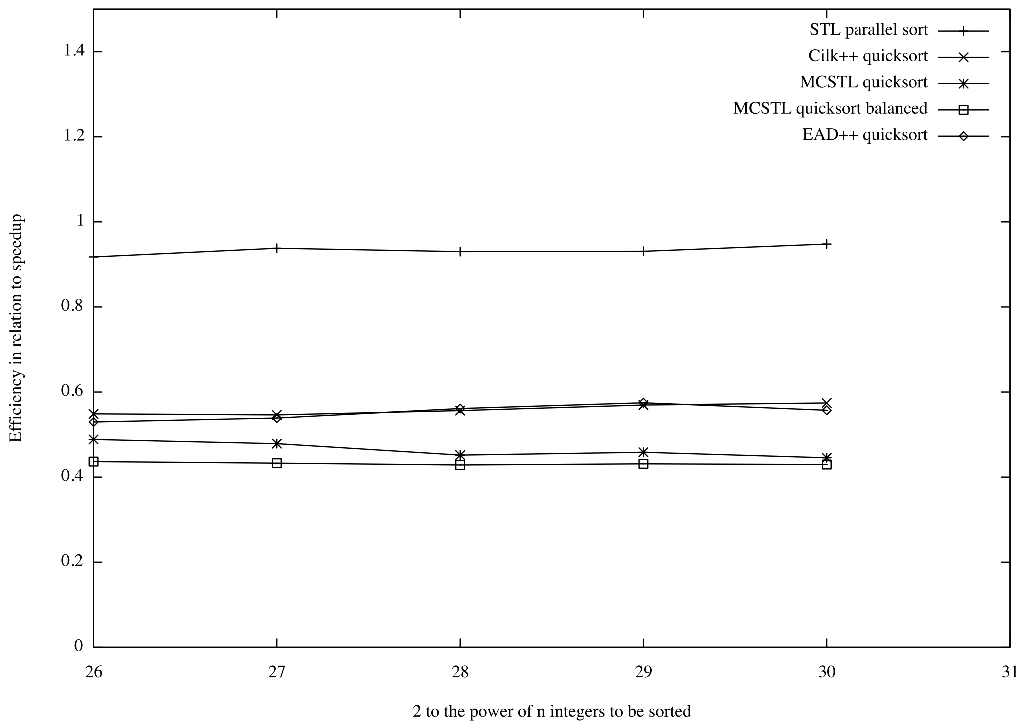
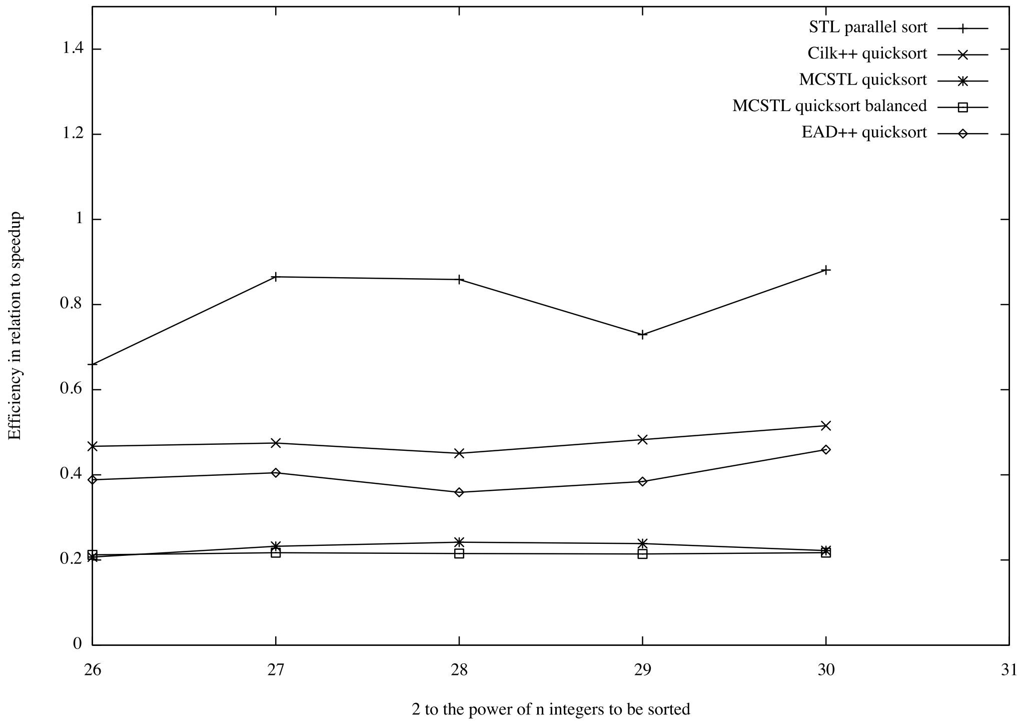


Figure C.30: Quicksort efficiency on tetra-octacore with 2 and 4 threads.



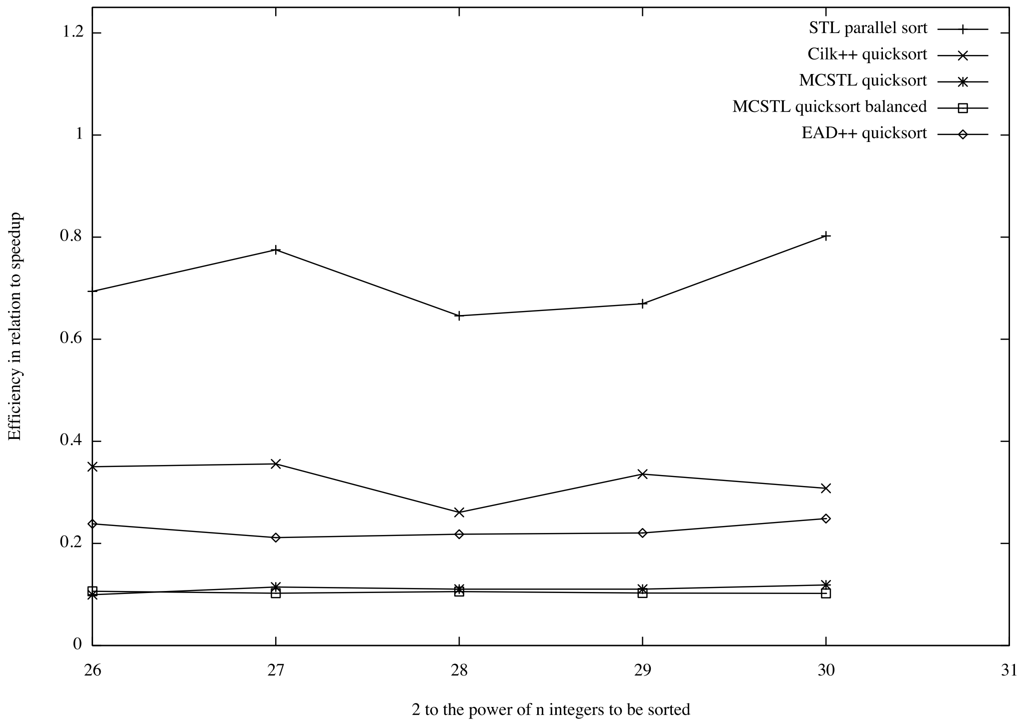


Figure C.31: Quicksort efficiency on tetra-octacore with 8 and 16 threads.

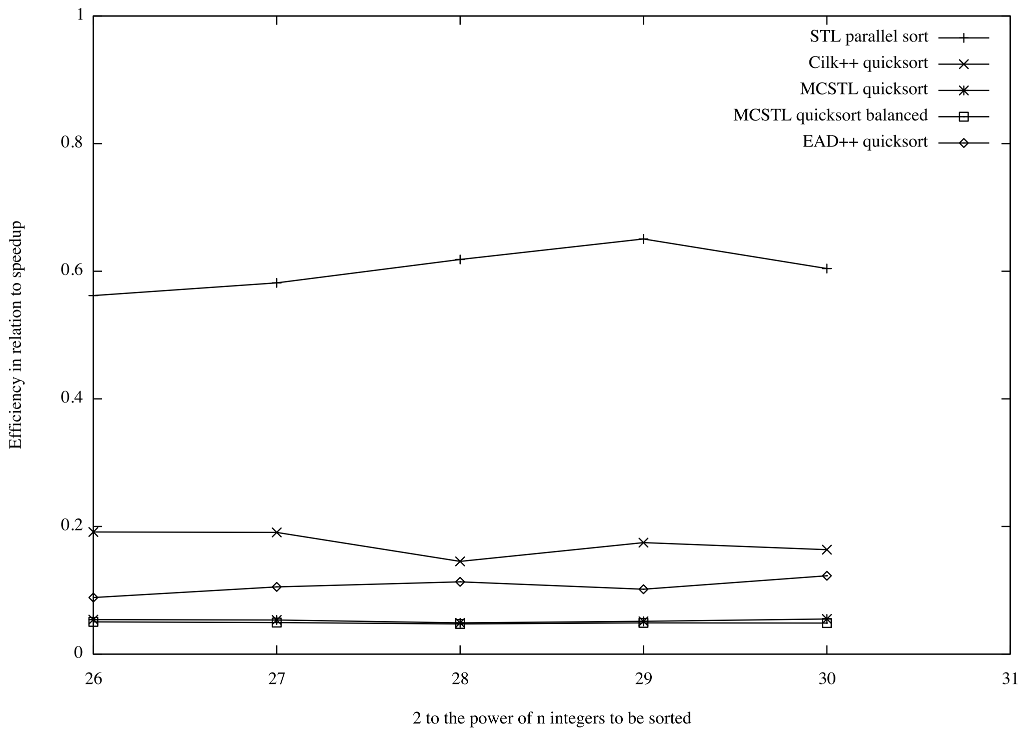
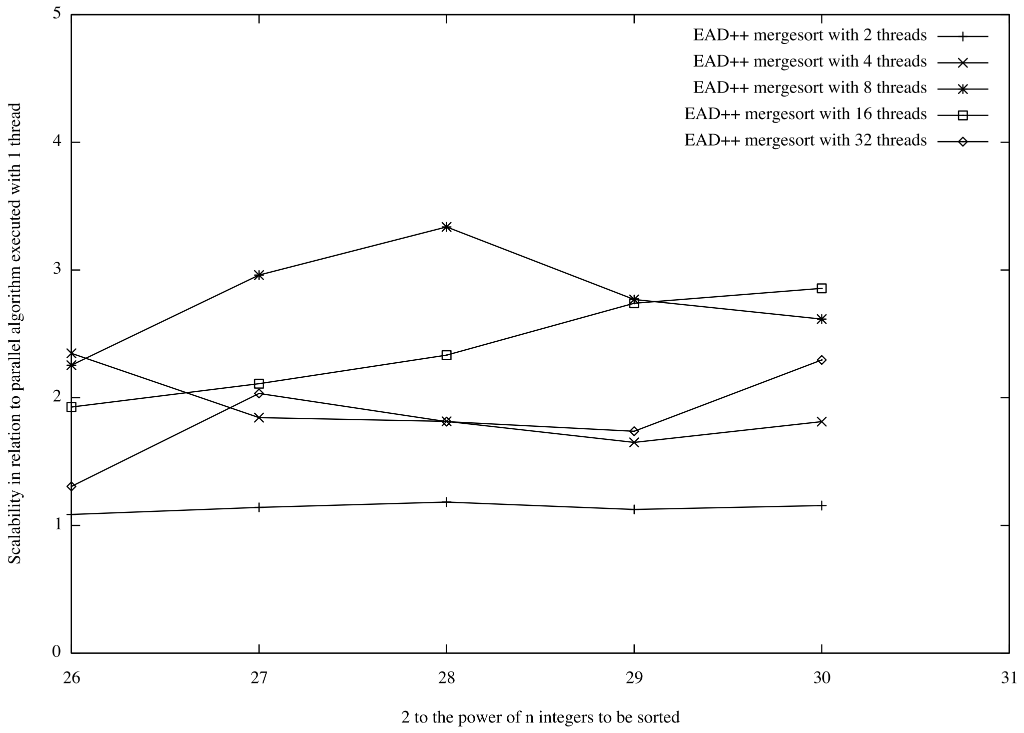


Figure C.32: Quicksort efficiency on tetra-octacore with 32 threads.



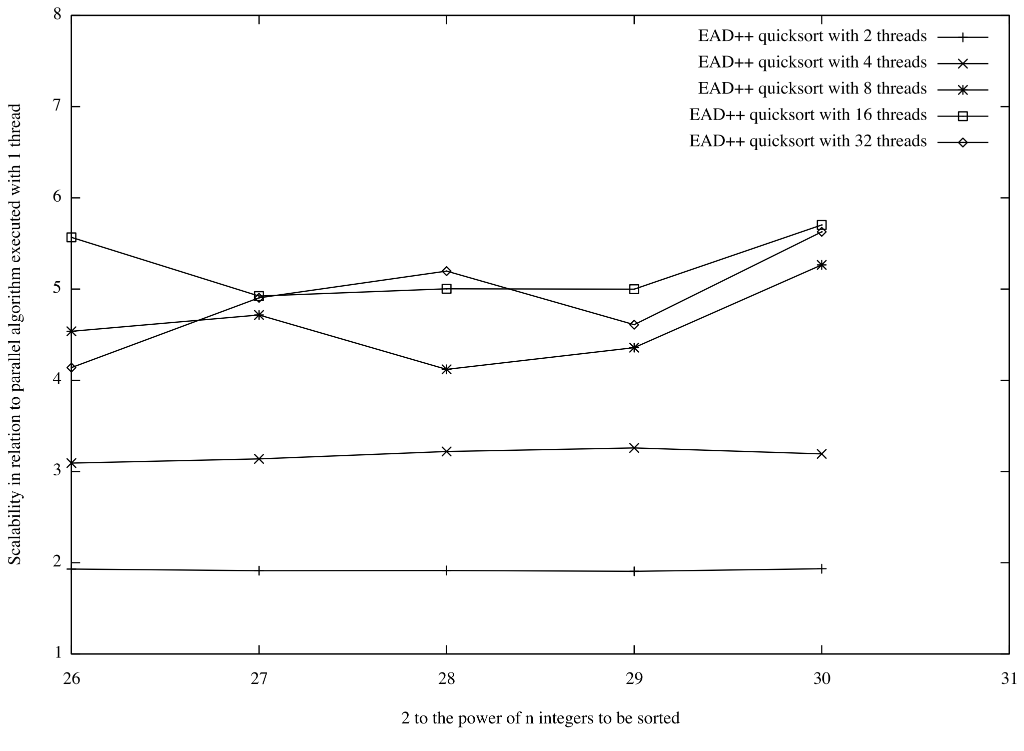
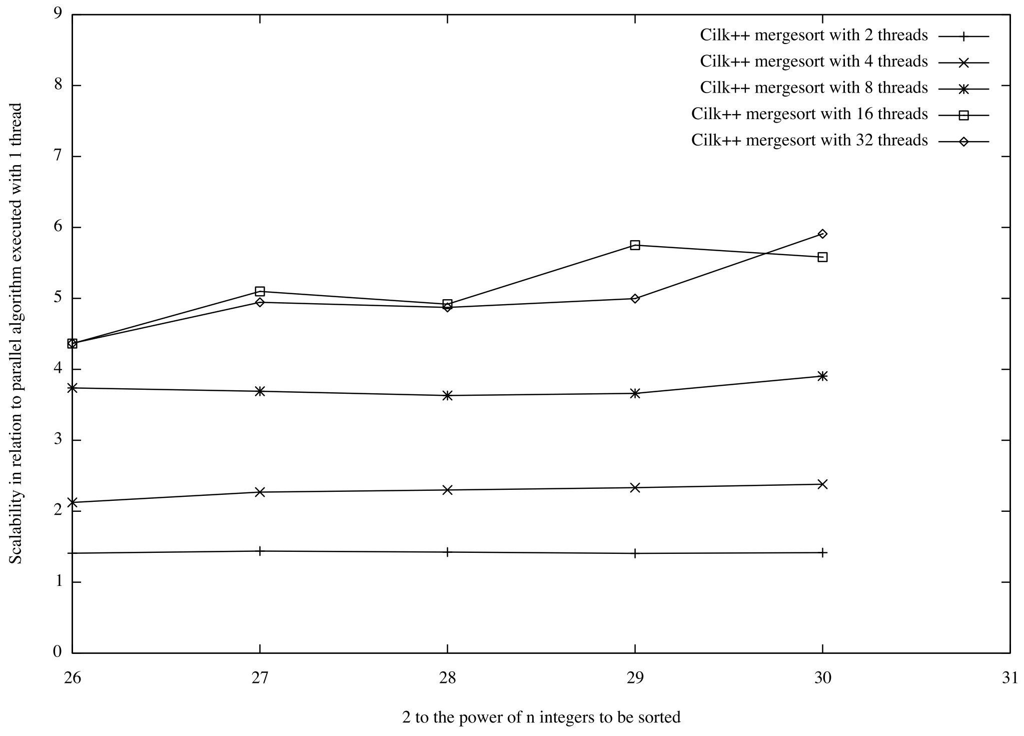


Figure C.33: Scalability of EAD++ mergesort and quicksort on tetra-octacore.



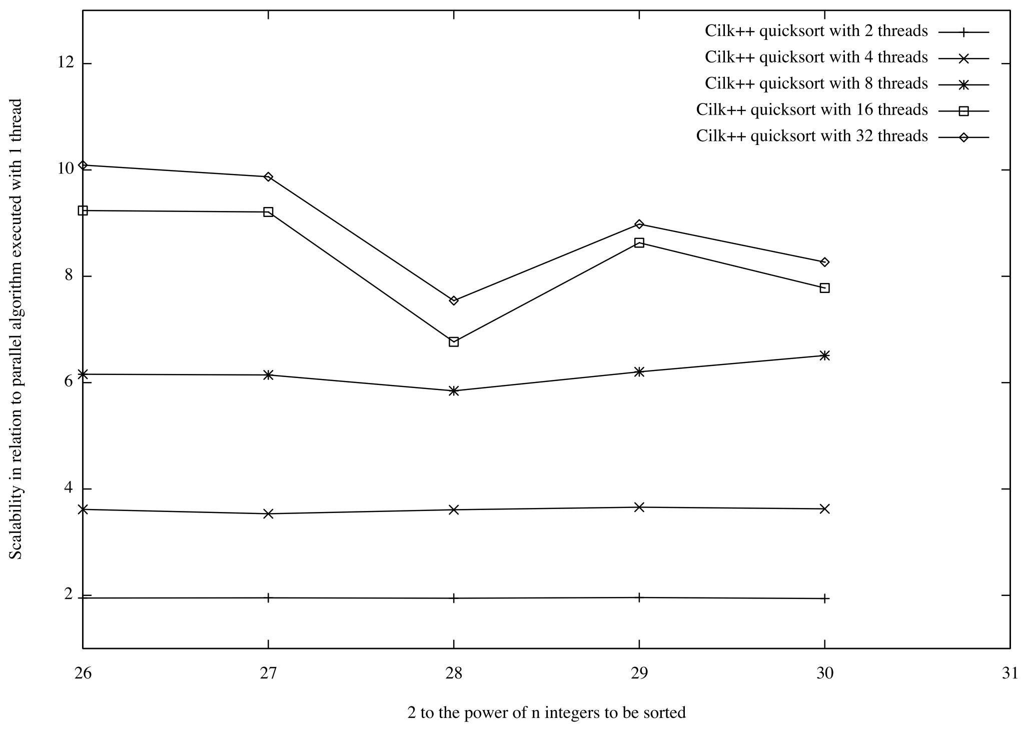
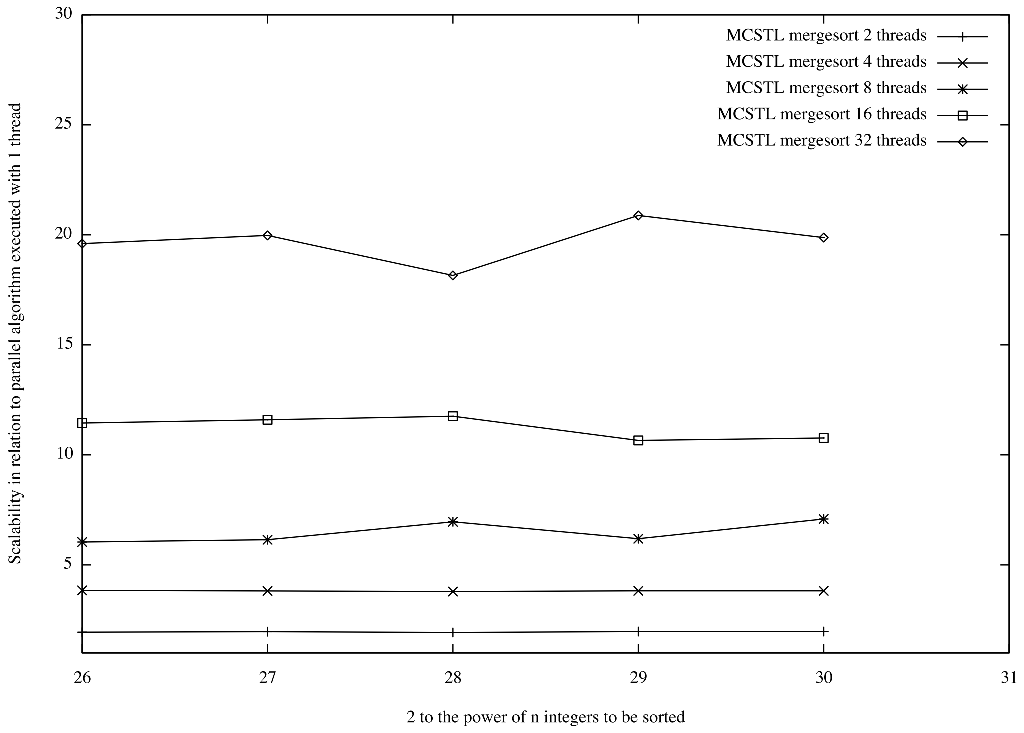


Figure C.34: Scalability of Cilk++ mergesort and quicksort on tetra-octacore.



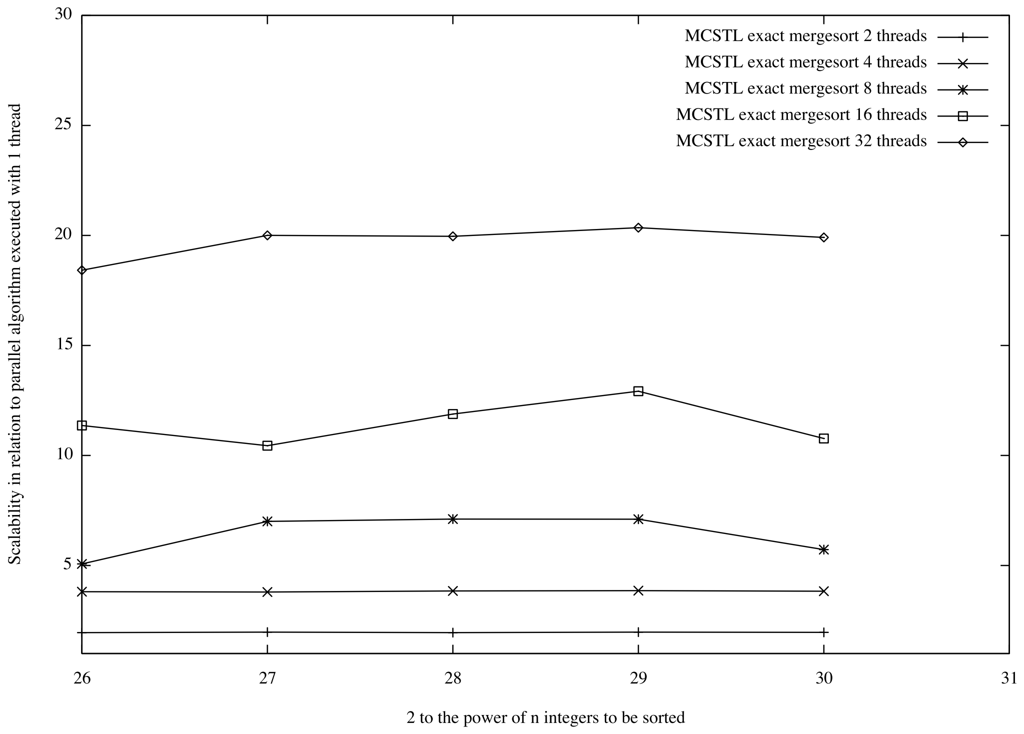
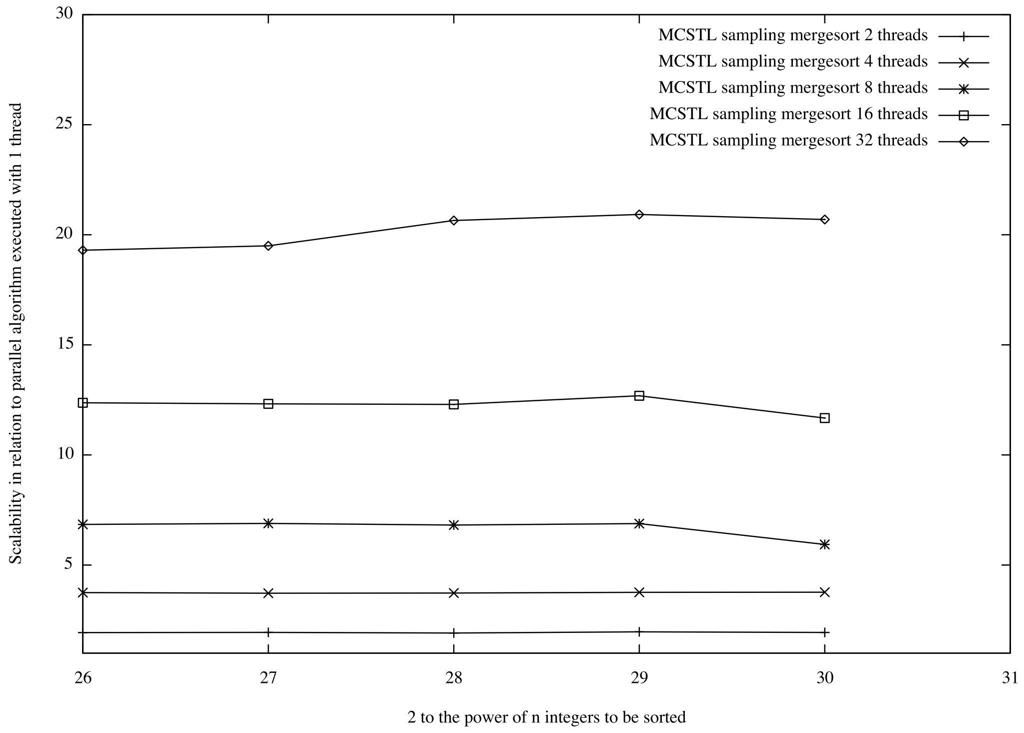


Figure C.35: Scalability of MCSTL mergesort and sort on tetra-octacore.



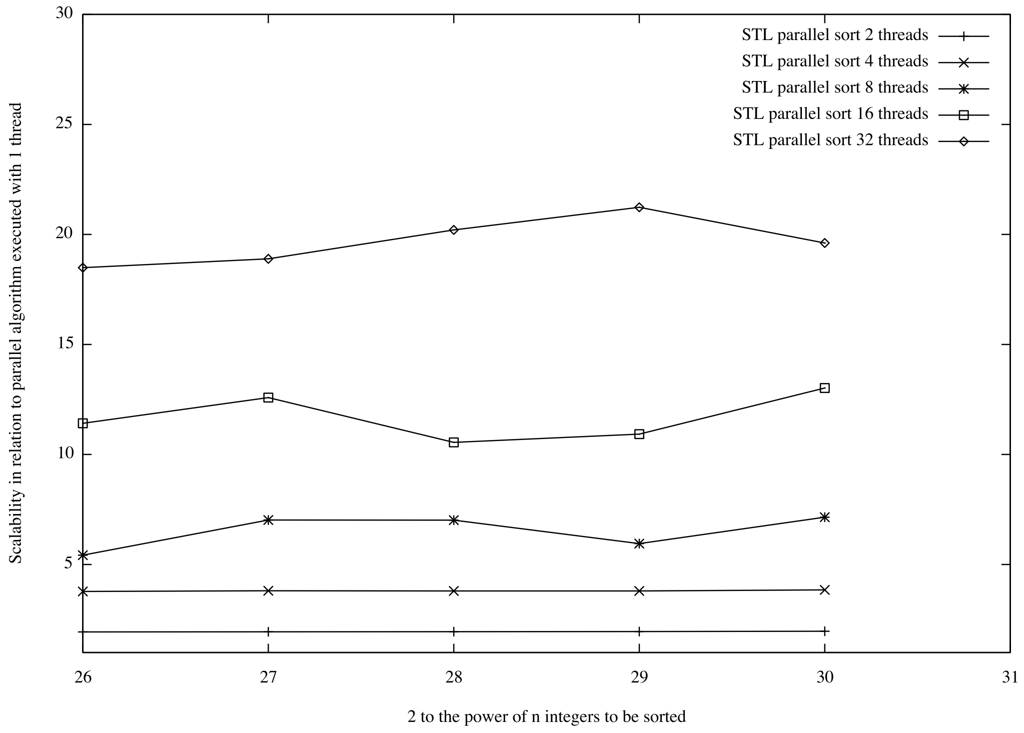
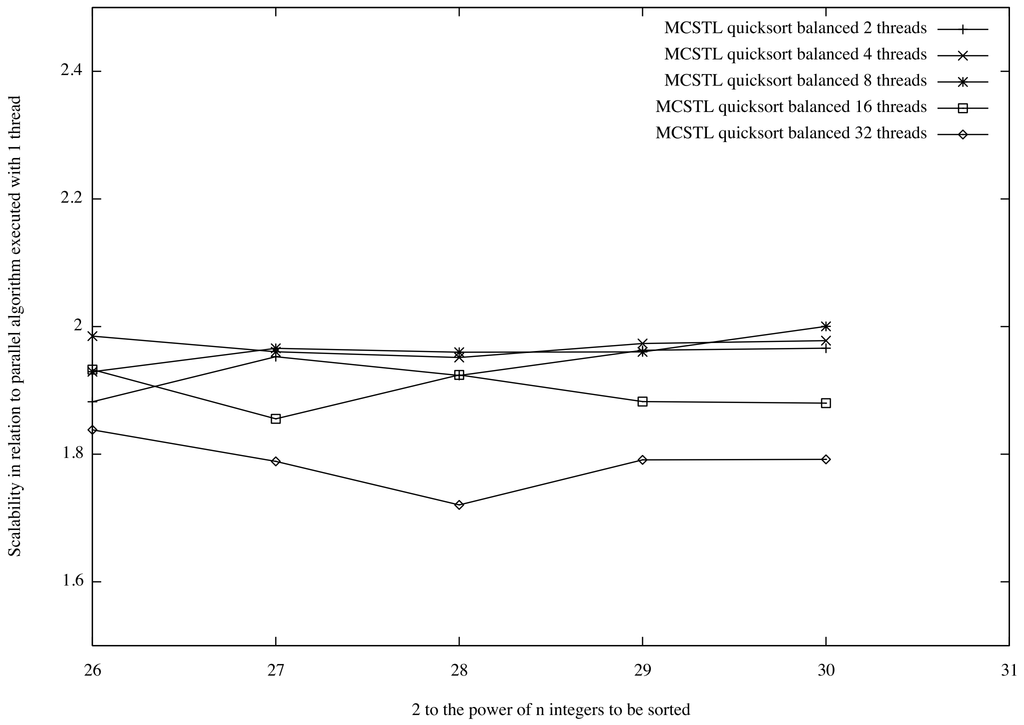


Figure C.36: Scalability of MCSTL mergesort and sort on tetra-octacore.



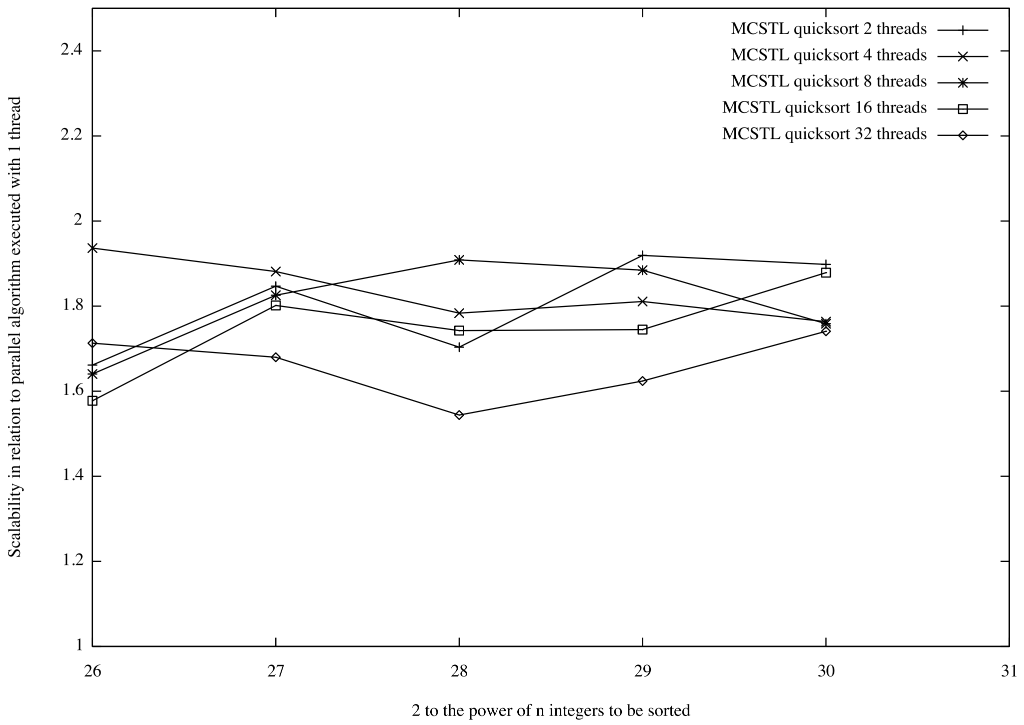


Figure C.37: Scalability of MCSTL quicksort and balanced quicksort on tetra-octacore.