Scalability Validation of Parallel Sorting Algorithms

Validierung der Skalierbarkeit von parallelen Sortieralgorithmen

Bachelor-Thesis von Yannick Berens

Tag der Einreichung:

1. Gutachten: Prof. Dr. Felix Wolf
2. Gutachten: Sergei Shudler

TECHNISCHE UNIVERSITÄT DARMSTADT

Department of Computer Science
Laboratory for Parallel Programming

Scalability Validation of Parallel Sorting Algorithms
Validierung der Skalierbarkeit von parallelen Sortieralgorithmen

Vorgelegte Bachelor-Thesis von Yannick Berens

1. Gutachten: Prof. Dr. Felix Wolf
2. Gutachten: Sergei Shudler

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, Yannick Berens, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
Mir ist bekannt, dass im Falle eines Plagiats (§38 Abs.2 APB) ein Täuschungsversuch vorliegt, der dazu führt, dass die Arbeit mit 5,0 bewertet und damit ein Prüfungsversuch verbraucht wird. Abschlussarbeiten dürfen nur einmal wiederholt werden.
Bei der abgegebenen Thesis stimmen die schriftliche und die zur Archivierung eingereichte elektronische Fassung überein.

Darmstadt, den 16. Oktober 2017

(Yannick Berens)

Abstract

As single-core performance of processors is not improving significantly anymore, the computer industry is moving towards increasing the amount of cores per processor or, in the case of large-scale computers, by installing more processors per computer. Applications now need to scale in accordance with the increase of parallel computing power and software developers need to take advantage of this movement. And parallel sorting algorithms present basic building blocks for many complex applications.

In this thesis, we will validate the expected execution time complexities of five state-of-the-art parallel sorting algorithms, implemented in `C` using MPI for parallelization, by using a scalability validation framework based on Score-P and Extra-P. For each of the parallel sorting algorithms, we will create a performance model. These models will allow us to compare their scalability behaviour to the expectations. Furthermore, we will attempt to parallelize the local sorting step of the splitter-based parallel sorting algorithms via `C++11` threads, OpenMP tasks, and CUDA acceleration.

We construct the performance models, on which we base our evaluations, using uniformly randomly generated data. For most of the parallel sorting algorithms, we show that the given expectations match the created models. We will discuss any other discrepancies in detail.

Zusammenfassung

Da die Leistung einzelner Prozessoren sich nicht mehr wesentlich verbessert, setzt die Computerindustrie immer mehr darauf die Anzahl der Prozessorkerne pro Prozessor zu erhöhen, oder im Falle von Großrechnern, die Prozessoren pro System. Anwendungen müssen nun entsprechend der Zunahme der parallelen Rechenleistung skaliert werden und Softwareentwickler müssen sich dieser Bewegung anschließen. Parallele Sortieralgorithmen stellen hierbei grundlegende Bausteine für viele komplexe Anwendungen dar.

In dieser Arbeit werden wir die Komplexitätserwartungen von fünf parallelen Sortieralgorithmen validieren, die in C und MPI implementiert wurden, indem ein Skalierbarkeitsvalidierungsframework basierend auf Score-P und Extra-P angewendet wird. Für jeden der parallelen Sortieralgorithmen werden wir ein Leistungsmodell erstellen. Diese Modelle erlauben es uns ihr Skalierungsverhalten mit den Erwartungen zu vergleichen. Außerdem werden wir versuchen, den lokalen Sortierschritt der Splitterbasierten parallelen Sortieralgorithmen mithilfe von C++11 Threads, OpenMP Tasks und CUDA Beschleunigung zu parallelisieren.

Wir konstruieren die Leistungsmodelle, auf denen wir unsere Auswertungen basieren, indem wir einheitlich zufällig erzeugte Daten verwenden. Für einige parallelen Sortieralgorithmen zeigen wir, dass die gegebenen Erwartungen mit den erstellten Modellen übereinstimmen. Wir werden weitere Diskrepanzen im Detail besprechen.

Contents

1 Introduction

Sorting is at the core of most applications - notable examples include search engines, file explorers, email clients and even address books. As a result, the field of sorting has been studied heavily and is featured in a large amount of literature. With the need to sort an ever-growing amount of data, developers are pressed to further their understanding and invent new methods to improve performance. But this movement has long outpaced the increase in single core performance. And as we can see in the movement in the consumer computer industry towards parallelism in the form of multi-core processors, we need to consider the sorting problem under the consideration of parallel computing. Especially the field of High-Performance Computing (HPC) sees increasing availability of parallelism thanks to the large array of processors installed. This calls for solutions that abide by the constraints of distributed memory and limited communication bandwidth that come with this environment.

But not only sequential sorting was studied extensively. There also already exists a huge amount of literature on parallel sorting algorithms. A subset of these is tailored to the structure of HPC applications. Using the 'Message Passing Interface' (MPI) for the parallelization, we can implement these algorithms on a multitude of architectures.

Most notably there are splitter-based parallel sorting algorithms. Splitters are a sorted list of elements that partition the process local data in a way that each partition can then be sent to another process. After a necessary merging step this will result in a globally sorted array. This way, large amount of data is only sent once and the communication steps are kept to a minimum. State of the art splitter-based algorithms include sample sort [1], that constructs its splitters from samples of each process. Another popular algorithm is histogram sort [2] that finds splitters in a way where the data distribution after the sorting is kept more even across all processes. More recently an algorithm called exact splitting sort [3] has been proposed that will keep the distribution of data across all processes the same after the sorting step and before.

There are also more sophisticated algorithms that are based on different fundamental ideas of sorting. Parallel radix sort [4] is a non-comparison sorting algorithm. It sorts the data by looking at the binary representation of each element and can this way determine where an element belongs. There are also special case algorithms that operate on only very specific parameters. For example, mini sort [5] assumes minimal data per process.

Naturally, all of the above-mentioned parallel sorting algorithms will perform differently depending on input and amount of processes as well as the underlying system. To understand the behaviour of these parallel sorting algorithms, you would need to run extensive benchmarks on a highly scalable system. This takes expensive computing power and, more importantly, time. As these benchmarks are closely linked to run time parameters, it is also not easily possible to draw conclusions to the general behaviour of the observed application.

Performance modeling tries to fix this problem by constructing an expression that will define the algorithms run time behaviour on a given parameter. For parallel applications, this method can model its scalability in the form of a function of the process count p. For example, imagine two functions f_1 and f_2 that model the run time complexity of two different algorithms. For $f_1 = 4 + p$ and $f_2 = 125 + \log_2(p)$[1], f_1 will perform better than f_2 for a process count less than 128. f_2 shows a better scalability behaviour.

We differentiate between analytical and empirical performance modeling. In analytical performance modeling, experts analyze the underlying code of a given algorithm by hand. Since

[1] Going forward log will refer to the logarithm of base 2 $\log_2$.

this method is time intensive, they will usually focus on only the most time-critical functions. But to create an accurate model, which is of great importance to find scalability bugs and efficiently analyze the algorithm, every function will need to be considered. The resulting expression is usually written in big O notation (Landau notation) which represents the upper bound complexity. Empirical performance modeling, on the other hand, can automatically create models from measurements. As such it is well suited to deduct the scalability behaviour of parallel sorting algorithms.

Extra-P [6] is a tool developed at Technische Universität Darmstadt to automatically generate performance models. We will use Score-P [7] to create a set of measurement that is provided to Extra-P. The measurements are extrapolated to create a model of the scalability behaviour on even higher scaling systems.

Given the added complexity that comes with parallelism and the usage of analytical performance modeling, we can not be sure of the accuracy of the expectations provided with the parallel sorting algorithms. In this thesis, we make the following contributions:

- *libparsort* library — we will consolidate the five state-of-the-art parallel sorting algorithms (sample sort, histogram sort, exact splitting sort, radix sort and mini sort) into one coherent and usable library using C, C++, and MPI. This includes documentation of the functions provided by the library.

- Validating expectations — by making use of a scalability validation framework, we can easily benchmark our library and create performance models. It utilizes Score-P for the instrumentation and Extra-P for the modeling.

- Experimental hybrid solutions — by extending the splitter-based parallel sorting algorithms in our library with multithreaded code using C++11 threads [8] and OpenMP tasks [9] and accelerated code using CUDA [10], we will analyze the benefits of these solutions. The scalability validation framework can be used here to compare the performance of the new code with the model of the old algorithm.

The remainder of this document is structured as follows: In Chapter 2, we summarize related works and put them in relation to our own approach. In Chapter 3, we provide the necessary background and fundamentals regarding parallel computing, sorting in sequence and in parallel, and the tools behind the scalability validation framework - Score-P and Extra-P. Chapter 4 describes the approach as outlined in the contributions above, followed by an evaluation in Chapter 5. Lastly, we will conclude our findings and discuss potential future work in Chapter 6.

2 Related Work

The contributions of this thesis combine the fields of parallel sorting, automatic generation of empirical performance models and the usage of multiple parallel computing approaches such as MPI, OpenMP, C++11 threads, and CUDA in one application. With this knowledge, we will validate the complexity expectations of state-of-the-art parallel sorting algorithms. Furthermore, we will attempt to optimize these sorting algorithms by multithreading and acceleration. In this chapter, we will explore existing literature related to this process, while keeping the separate topics in mind.

This work is heavily influenced by an unpublished paper by Siebert et al. [11] that analyzed dynamic load balancing in parallel sorting algorithms. We were provided the paper itself and a code base consisting of implementations of the parallel sorting algorithms discussed in it. We would like to thank the authors as their work allowed us to create this thesis in the first place.

Of the approaches to the computer-aided analysis of parallel sorting algorithms known to us, the one by Blelloch et al. [12] is the oldest we found. It uses a fine grain approach to analyzing the algorithms, by benchmarking primitive operations. These benchmarks were then applied to parallel radix sort and sample sort by counting the kind and amount of primitive operations each one calls. This approach is highly accurate, but ultimately irrelevant for our use case. Extra-P offers a way to quickly and automatically generate performance models that can accurately express the scalability behaviour of a given algorithm.

In 1996 Amato et al. [13] compared parallel sorting algorithms on different architectures. The parallel sorting algorithms include sample sort and a parallel radix sort. Their novel method of constructing the expected complexities with two components - local computation L and interprocessor communication C - allowed them to accurately predict scalability behaviour of these algorithms. But more importantly, they tested the parallel sorting algorithms on a distributed memory system.

In 2015 Shudler et al. [14] used Score-P and Extra-P to create a validation framework. They applied a case study about MPI collective functions to the framework and validated the resulting models to the expectations. This approach can easily be applied to the validation of parallel sorting algorithms. Furthermore, the results of the MPI operations can be used to understand the scalability behaviour of the parallel sorting algorithms more closely.

White et al. [15] proposed a hybrid CUDA-MPI solution for parallel sorting in 2012. Based on their work another paper with a similar subject has been released in 2013 by Shenghui et al. [16]. On a cluster of GPUs, each GPU has an array of data that is sorted using parallel bitonic sort implemented using CUDA. Afterwards, they use MPI to merge the data. Their experiments show optimistic improvements in run time. Our approach to hybrid parallel sorting also includes a variation of a CUDA-MPI solution. But as we will need to copy large amounts of data between CPU and GPU at least two times in our implementation, it is to be seen if we will observe similar results.

A hybrid parallel merge sort using MPI and OpenMP has been proposed and evaluated by Radenski [17]. In his work, he compares a shared memory implementation with OpenMP, a distributed memory implementation and a hybrid implementation of merge sort against each other. In the evaluation, we can observe significant speed ups with the hybrid solution.

3 Background

Before presenting our approach in Chapter 4, we discuss in this chapter all the fundamentals of parallel sorting and the validation process. Firstly we discuss the basics of comparison-based sorting, how it relates to parallel computing and what parallel sorting algorithms look like. Then all the specific algorithms used in this thesis are outlined. And lastly, we focus on Score-P and Extra-P.

3.1 Sorting Problem

Sorting has been around long before the electronic age, but where a person has limited possibilities to sort, a computer can take very different approaches. The general problem definition for sorting is defined as follows:

Input:	A sequence of n items $(x_1, x_2, \ldots, x_n)$, and a relational operator $\leq$[1]that specifies an order on these items.
Output:	A permutation (reordering) $(y_1, y_2, \ldots, y_n)$ of the input sequence such that $y_1 \leq y_2 \leq \ldots \leq y_n$.

There exist many solutions to the problem of sorting in computing, as it was studied for more than sixty years. The most popular solutions are based on comparison and build upon a strategy called 'divide and conquer'. Here the problem size n is repeatably divided into two partitions. This creates a binary tree structure which is eventually merged from the leaves to the root. Often used algorithms include quick sort and merge sort that both work on this process. To understand the idea, let's look at our implementation of merge sort in Algorithm 1.

Algorithm 1 Merge Sort

```
Input list $A$ with $|A| = n$, a lower bound $lo$ and a higher bound $hi$
1: if $lo + 1 < hi$ then
2:     mid = $[(lo + hi)/2]$
3:     mergesort($A, lo, mid$)
4:     mergesort($A, mid, high$)
5:     merge($A, lo, mid, hi$)
6: end if
7: return $A$
```

By providing the merge sort function with a lower and higher bound for the data that should be observed, we can recursively call this function with only a subset of the initial input. This step is repeated until we are left with only one element. Now a sorting merge algorithm is used to sort the partitions. This way of dividing data can already be applied to parallel computing by simply running each subset on a different processing unit.

[1] Sorting is not limited to the relational operator $\leq$, but will only be discussed for ascending order in this thesis. Though the underlying functionality would not change in a drastic way if another relational operator was chosen.

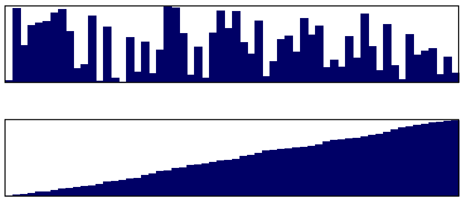


Figure 3.1: Visual representation of the sequential sorting problem definition with $n = 60$ keys.

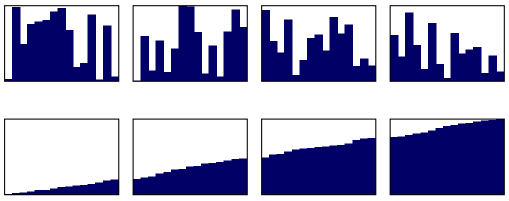


Figure 3.2: Visual representation of the parallel sorting problem definition for distributed memory(MPI) with $p = 4$ processes and $n = 15 + 15 + 15 + 15$ keys.

3.2 Parallel Computing

There are a lot of different approaches to creating parallel applications. The most basic idea is to run parts of your code in parallel by scheduling them to threads. Threads are usually managed by the operating system. Recently C++ has made strives towards an exception-safe and hardware-agnostic implementation of threads in C++11.

The ability to parallelize your code has not always been of high priority. As such a method was devised to parallelize your application without the need to rewrite large parts of the underlying code. OpenMP works on this paradigm and provides parallelism concepts directives in the form of `#pragma` statements.

The device that is capable of the most parallelization in your everyday computer is not the CPU though, but the GPU. With the help of NVIDIA's CUDA, you can program your graphics card for large Single Instruction Multiple Data (SIMD) applications even outside of graphical computing. The downside of working with both the CPU and GPU is the fact that each has their own memory. This means we will have to copy data between the two devices.

The last method of parallel computing we will present and ultimately use the most in this thesis is MPI. The "Message Passing Interface" is a library, that allows communication between processes in the form of messages that hold data. It is especially suitable for large-scale computers and cluster, as it operates on the assumption that every process has its own separate memory. Each of the p processes is given a *rank* with which the processes can communicate with each other. The *rank* is an unsigned integer from $\{0, \ldots, p-1\}$. MPI also provides collective functions that allow for more advanced communication, such as broadcasting data to all other processes at once.

3.3 Parallel Sorting Algorithms

The general problem definition for parallel sorting with distributed memory is defined as follows:

Input: A sequence of n items $(x_1, x_2, \dots, x_n)$ over p processes with $n_i = n/p$ items per process, and a relational operator $\leq$ that specifies an order on these items.

Output: A permutation (reordering) $(y_1, y_2, \dots, y_n)$ of the input sequence such that for the local data z $z_1 \leq z_2 \leq \dots \leq z_n$ on each process i and $z_i[n_i n - 1] \leq z_{i+1}[0]$ for $i \in \{0, \dots, p-2\}$.

As opposed to the sequential case seen in Figure 3.1, the initial situation has changed in a way where the known sequential algorithms can not trivially be fit to the parallel sorting problem. As seen in Figure 3.2 the first process will need to access or receive data from the other processes.

As such our algorithms are designed in a way that complements the design of MPI. The biggest performance bottleneck is the communication between processes, so this will be kept to a minimum. Another thing to consider is the fact that there are specific syncing points throughout the code, so the performance can be largely bounded by only the slowest process. That is why the parallel sorting algorithms try to achieve evenly sized arrays distributed over all p processes.

As part of our library *libparsort*, we made a selection of state-of-the-art parallel sorting algorithms. These include splitter-based algorithms such as sample sort, histogram sort, and exact splitting sort. Here $p-1$ splitter elements are collected and used to partition the data of each process such that for $i \in \{0, \dots, p-2\}$ partition i can be sent to the process of rank i. The library also includes parallel radix sort, a non-comparison algorithm and a special case algorithm for minimal data per process called mini sort.

3.3.1 Sample Sort

Sample sort presents itself as a rather simple implementation of the splitter based approach. It tries to achieve even distribution by sampling the data of the array to construct its splitters. But it can not ensure that the data is sufficiently evenly distributed over the arrays because of the data dependency of the sampling process.

Algorithm 2 Sample Sort

```
Input list $A$ with $|A| = n_i$ on process $i$
1: Sampling                                          ▷ Local sort depending on variation
2: $X \leftarrow \texttt{gather}((x_i)_{i\in\{1,\dots,k\}})$ on *root*
3: *root* **only:** $X \leftarrow \text{sort}(X)$
4: *root* **only:** $\forall i \in \{1,\dots,p-1\} : s_i \leftarrow X[i \cdot k]$
5: $\forall i \in \{1,\dots,p-1\} : s_i \leftarrow \texttt{broadcast}(s_i)$ from *root*
6: $(A_i)_{i\in\{1,\dots,p\}} \leftarrow \text{partition}(A, (s_i)_{i\in\{1,\dots,p-1\}})$
7: $(B_i)_{i\in\{1,\dots,p\}} \leftarrow \texttt{alltoall}((A_i)_{i\in\{1,\dots,p\}})$
8: **return** $B \leftarrow \text{sort}(\text{concat}((B_i)_{i\in\{1,\dots,p\}}))$
```

$p-1$ samples from each process are collected on the *root* process[2]. These $p \cdot (p-1)$ samples are sorted and $p-1$ splitters are chosen from them equidistantly. They are then distributed

[2] While the *root* can be arbitrarily defined by the developer, it usually refers to the process with *rank* 0, as it is sure to exist at any number of processes.

to all processes, where its local data of n_i elements is partitioned accordingly. Afterwards, the partitions are sent to each corresponding process and merge via sequential sort. This will keep communication data to a minimum.

There are a few different approaches to how the splitters are sampled. Most notably there is regular sampling, where the samples are chosen equidistantly. This requires the data on each process to be sorted sequentially in advance. There is also random sampling, where the samples are chosen randomly, so the local data does not need to be sorted. In this case, the partitioning needs to traverse the whole array and can not take advantage of a binary search approach. Either way, the splitters are always chosen equidistantly from all the samples.

3.3.2 Histogram Sort

Histogram goes one step further to ensure sufficiently evenly distributed data across the processes. Besides the array (and size), the algorithm also asks for a threshold value $q \in [0;1]$ such that $n_i \in [(1-q)\cdot\frac{n}{p};(1+q)\cdot\frac{n}{p}]$ for $i \in \{0,\ldots,p-2\}$. This means it can successfully ensure a sufficiently even distribution. But there is no guarantee that the algorithm finishes as the construction of the splitters is only using data from the *root* process.[3]

Algorithm 3 Histogram Sort

```
Input list $A$ with $|A| = n_i$ on process $i$
 1: $A \leftarrow$ sort($A$)
 2: $(t_i)_{i\in\{1,\ldots,p\}} \leftarrow$ allgather(scan($n_i$, +))
 3: loop
 4:     root only: $\forall i \in \{1,\ldots,p-1\} : s_i \leftarrow A[j]$
 5:         $j$ from interpolation between known counts.
 6:     broadcast($i$) from root
 7:     $c_i \leftarrow |\{a \in A | a \le s_i\}$
 8:     $C_i \leftarrow$ allreduce($c_i$, +)
 9:     if $\forall i \in \{1,\ldots,p\} : |t_i - C_i| < \epsilon n_i$ then
10:         break
11:     end if
12: end loop
13: $(A_i)_{i\in\{1,\ldots,p\}} \leftarrow$ partition($A$, $(s_i)_{i\in\{1,\ldots,p-1\}}$)
14: $(B_i)_{i\in\{1,\ldots,p\}} \leftarrow$ alltoall($(A_i)_{i\in\{1,\ldots,p\}}$)
15: return $B \leftarrow$ merge($B_i$)
```

Each process sorts its data array locally. Then $p-1$ samples are taken only from the *root* process. The samples are sent to all p processes where they analyze how many elements are smaller than each sample. These values are collected on the *root* process and sum reduced creating a histogram C. Now for each sample s_i with $i \in \{1,\ldots,p-1\}$ there exists a histogram entry C_i with the same index. If the sample is in range of the ideal splitter position where for s_i, $i*(1+q)*\frac{n}{p} \le C_i \le i*(1-q)*\frac{n}{p}$ it will be used as a splitter. If a sample is not a satisfactory splitter, a new sample is collected. Once all splitters have been determined, the partitions are distributed to the corresponding processes followed by a local sequential sort.

[3] Some other specifications may call for the splitters to be taken from all the processes. Another approach by Solomonik et al. constructs a search space with bounds based on the range of the input over all processes, and shrinking it with each necessary iteration [18].

3.3.3 Exact Splitting Sort

Exact splitting sort is a rather new splitter-based parallel sorting algorithm. It is the last of the splitter-based parallel sorting algorithms and as the name implies, tries to find splitters that would partition the data such that n_i after sorting equals n_i before sorting. Assuming an even distribution before calling the algorithm this would mean $n_i = \frac{n}{p}$ - an ideal distribution.

Algorithm 4 Exact Splitting Sort

```
Input list $A$ with $|A| = n_i$ on process $i$
1: $A \leftarrow \text{sort}(A)$
2: global min & global max creates a search space
3: splitter finding over the search space using binary search with global median
4: $(A_i)_{i\in\{1,\dots,p\}} \leftarrow \text{partition}(A, (s_i)_{i\in\{1,\dots,p-1\}})$
5: $(B_i)_{i\in\{1,\dots,p\}} \leftarrow \texttt{alltoall}((A_i)_{i\in\{1,\dots,p\}})$
6: return $B \leftarrow \text{p-way-merge}(\text{concat}((B_i)_{i\in\{1,\dots,p\}}))$
```

As with each of the splitter-based algorithms, the first step is a local sorting step. Then local minima and maxima are created based on the smallest and largest elements. From these values a global minimum and maximum can be derived by using an MPI reduction function. The $p-1$ splitters are then found within this search space via binary search. Using the global median, the search space in split into two parts and we evaluate the splitter by counting the elements less than the splitter and comparing it to n_i. This step is repeated until all splitters have been found. As with sample sort and histogram sort before, partitions are created and distributed to the corresponding process. But instead of local sequential sort, a more sophisticated p-way-merge algorithm is used.

3.3.4 Radix Sort

Radix Sort is a non-comparison sorting algorithm that sorts elements based on its binary presentation. This way it can overcome the lower bound complexity of traditional comparison-based sorting algorithms of $n \cdot \log n$. Due to the way it works it is limited to integer- and float-like data types and its performance is dependant on the number of bits and number of digits of the given data.

Algorithm 5 Radix Sort

```
Input list $A$ with $|A| = n_i$ on process $i$
1: for $1 \le r \le b$ by $k$ do
2:     $(A_i)i \in \{1, \dots, 2^k\} \leftarrow \text{partition}(A)$ by bits $r, \dots, r+k-1$
3:     for $1 \le i \le 2^k$ do
4:         $\forall$ processes: append($A_i$)
5:     end for
6: end for
7: return $A$
```

The algorithm sorts each element of b bits one digit of k bits at a time. This is done using counting sort. This way the data is locally partitioned into 2^k blocks. Looking at the distributed memory of all processes as a global memory, the blocks are put into the global memory. This creates a globally sorted sequence.

3.3.5 Mini Sort

Mini sort is a rather new parallel sorting algorithm. It assumes the data on each process to be minimal. In our case this means the local array is of size 1, resulting in $n = p$.

Algorithm 6 Mini Sort Sequential

```
Input  element $x_i$ on process $i$
1: $root$ only: $X \leftarrow \texttt{gather}(x_i)$
2: $root$ only: $X \leftarrow \text{sort}(X)$
3: $y \leftarrow \text{send}(X[i])$ to rank $i$
4: return $y$
```

Algorithm 7 Mini Sort Counting

```
Input  element $x_i$ on process $i$
1: $X \leftarrow \texttt{allgather}(x_i)$
2: dest$= -1$
3: $\forall X_i : i \in \{1, \dots, p\} \leq x_i \rightarrow$ dest$++$
4: $y \leftarrow \text{send}(x_i)$ to dest
5: return $y$
```

As with sample sort there are different variations of this algorithm. In the naive approach, each process sends its data to the *root* process, where it is sorted sequentially. Then each element $x_i i \in \{0, \dots, p\}$ on the *root* is sent to process i. Another approach calls for each process to send its data to all p processes where a counting algorithm is used on the resulting sequence to determine the destination of its original data. It works very similarly to the naive mini sort algorithm.

A more sophisticated mini sort algorithm uses a median-of-three reduction scheme coupled with a tree recursion model to find the global median. This value is then used as a pivot to partition the single element data on each process into three groups; less than the pivot, equal to the pivot and greater than the pivot. Using vector structures and reduction sum functions, the destination process can be calculated.

3.4 Performance Modelling

Complexity analysis is a big field of computer science. As we move into parallel space, determining accurate complexity expressions of a given parallel algorithm is getting more intricate because of the added complexity. But it can also be wildly influenced by the implementation of the algorithm, unoptimized code in the MPI library and experimentation setup. So as part of the validation process, we will be using empirical performance modeling to create complexity models of each parallel sorting algorithm. The scalability validation framework [14] we chose to use calls for the tools Score-P and Extra-P.

3.4.1 Score-P

"The Score-P measurement infrastructure is a highly scalable and easy-to-use tool suite for profiling, event tracing, and online analysis of HPC applications."[4]

[4] Taken from `http://www.vi-hps.org/projects/score-p/`(accessed on 2017-10-05).

We will use Score-P for its profiling capabilities. It consists of an instrumentation framework and run-time libraries, applicable to C/C++. Instrumentation describes the insertion of measurement functions into the source code. It does so by providing its own compiler wrapper that can wrap around a large collection of compilers. This way Score-P can automatically instrument the underlying code. You can also provide filters to adjust this automatic step or use manual instrumentation only. Either way, MPI calls will always be instrumented and measured.

For programs that were compiled with Score-P, measurements are automatically collected when the application is run. Metrics include visits and time as well as other metrics. This performance data is saved as a performance profile in the CUBE [19] data format. The format holds the data in 3 dimensions. The first one holds the performance metrics. The second dimension is a call tree. This way measurements for each function are stored and can later be analyzed in detail. The last dimension is a description of the system the application was executed on.

3.4.2 Extra-P

"Extra-P is an automatic performance-modeling tool that supports the user in the identification of scalability bugs."[5]

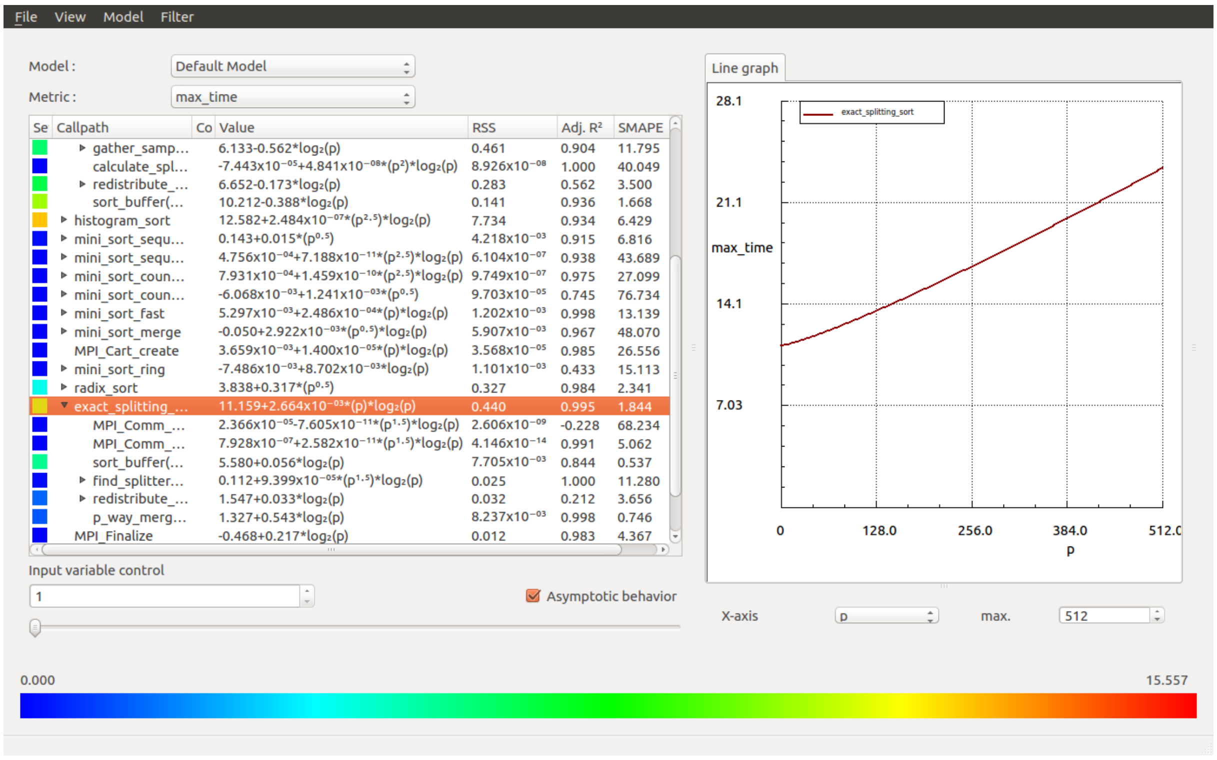
We will use Extra-P's performance modeling to express the scalability behaviour of parallel sorting algorithms. Extra-P takes a set of performance profiles that can be provided by Score-P. The performance profiles are collected by running the application multiple times with different configuration parameters. From that, a human-readable model in the form of a function is created, describing the scalability behaviour over these configuration parameters. For parallel sorting, this means running the program with different process counts. For accurate modeling, Extra-P needs measurements for at least five different parameter variations.

The resulting scaling models are assignments of the performance model normal form (PMNF), defined by the formula

$$f(p) = \sum_{n=1}^{k} c_k \cdot p^{i_k} \cdot \log(p)^{j_k}$$

Extra-P also introduces error metrics on the generated model. $\bar{R}^2$ is of range 0 to 1 and shows how well the constructed PMNF matches the underlying actual model, 1 being ideal. SMAPE, on the other hand, is a scale-independent, relative error metric presented by Reisert [20]. It uses the ratio of the difference between measured and predicted value to create an error metric of range 0 to 200, where 0 means no error at all.

[5] Taken from `http://www.scalasca.org/software/extra-p/download.html`(accessed on 2017-10-05).



Model : Default Model

Metric : max_time

Se	Callpath	Co	Value	RSS	Adj. R²	SMAPE
🟩	▸ gather_samp…		$6.133-0.562*\log_2(p)$	0.461	0.904	11.795
🟦	calculate_spl…		$-7.443\text{x}10^{-05}+4.841\text{x}10^{-08}*(p^2)*\log_2(p)$	$8.926\text{x}10^{-08}$	1.000	40.049
🟩	▸ redistribute_…		$6.652-0.173*\log_2(p)$	0.283	0.562	3.500
🟩	sort_buffer(…		$10.212-0.388*\log_2(p)$	0.141	0.936	1.668
🟧	▸ histogram_sort		$12.582+2.484\text{x}10^{-07}*(p^{2.5})*\log_2(p)$	7.734	0.934	6.429
🟦	▸ mini_sort_sequ…		$0.143+0.015*(p^{0.5})$	$4.218\text{x}10^{-03}$	0.915	6.816
🟦	▸ mini_sort_sequ…		$4.756\text{x}10^{-04}+7.188\text{x}10^{-11}*(p^{2.5})*\log_2(p)$	$6.104\text{x}10^{-07}$	0.938	43.689
🟦	▸ mini_sort_coun…		$7.931\text{x}10^{-04}+1.459\text{x}10^{-10}*(p^{2.5})*\log_2(p)$	$9.749\text{x}10^{-07}$	0.975	27.099
🟦	▸ mini_sort_coun…		$-6.068\text{x}10^{-03}+1.241\text{x}10^{-03}*(p^{0.5})$	$9.703\text{x}10^{-05}$	0.745	76.734
🟦	▸ mini_sort_fast		$5.297\text{x}10^{-03}+2.486\text{x}10^{-04}*(p)*\log_2(p)$	$1.202\text{x}10^{-03}$	0.998	13.139
🟦	▸ mini_sort_merge		$-0.050+2.922\text{x}10^{-03}*(p^{0.5})*\log_2(p)$	$5.907\text{x}10^{-03}$	0.967	48.070
🟦	MPI_Cart_create		$3.659\text{x}10^{-03}+1.400\text{x}10^{-05}*(p)*\log_2(p)$	$3.568\text{x}10^{-05}$	0.985	26.556
🟦	▸ mini_sort_ring		$-7.486\text{x}10^{-03}+8.702\text{x}10^{-03}*\log_2(p)$	$1.101\text{x}10^{-03}$	0.433	15.113
🟦	▸ radix_sort		$3.838+0.317*(p^{0.5})$	0.327	0.984	2.341
🟨	▾ exact_splitting_…		$11.159+2.664\text{x}10^{-03}*(p)*\log_2(p)$	0.440	0.995	1.844
🟦	MPI_Comm_…		$2.366\text{x}10^{-05}-7.605\text{x}10^{-11}*(p^{1.5})*\log_2(p)$	$2.606\text{x}10^{-09}$	-0.228	68.234
🟦	MPI_Comm_…		$7.928\text{x}10^{-07}+2.582\text{x}10^{-11}*(p^{1.5})*\log_2(p)$	$4.146\text{x}10^{-14}$	0.991	5.062
🟩	sort_buffer(…		$5.580+0.056*\log_2(p)$	$7.705\text{x}10^{-03}$	0.844	0.537
🟦	▸ find_splitter…		$0.112+9.399\text{x}10^{-05}*(p^{1.5})*\log_2(p)$	0.025	1.000	11.280
🟦	▸ redistribute_…		$1.547+0.033*\log_2(p)$	0.032	0.212	3.656
🟦	p_way_merg…		$1.327+0.543*\log_2(p)$	$8.237\text{x}10^{-03}$	0.998	0.746
🟦	MPI_Finalize		$-0.468+0.217*\log_2(p)$	0.012	0.983	4.367

Input variable control: 1

✓ Asymptotic behavior

X-axis p max. 512

0.000 15.557

Figure 3.3: Screenshot of Extra-P, showing data from the parallel sorting library

4 Approach

Before we can generate performance models of the scalability of parallel sorting algorithms, we will need to implement what we have discussed in Chapter 3. We will discuss the implementation of the library and our approach to the validation of the scalability behaviour of parallel sorting algorithms. Afterwards, we will present our solution to a parallel local sort for splitter-based algorithms.

4.1 Implementation

A large part of the algorithms was already implemented. But the code was scattered, partly incomplete or even nonfunctional and it lacked a cohesive direction. The code was written in C using OpenMPI [21] for the parallelization. We will use `gcc 4.9.4` and `OpenMPI 1.10.7` in conjunction with autotools to manage and compile the project.

The main objective of this phase was to create a complete and functional library consisting of all the algorithms ready to use. This phase was further split up into the following milestones:

- Write a simple test suite
- Condense all the code into a single unified library
- Make adjustments and fix bugs

In the first step a test program that would make use of the library and define test cases had to be created. Since we had no intention of using an uneven distribution, our test application only defines the number of elements each process should hold. Before any sorting takes place, we seed the random number generator in a unique way for each process. This seed is also used for any algorithm that depends on random numbers, such as random sample sort. Before each sorting algorithm is executed, we create uniformly random numbers to fill the local array.

Next up, we unified the the way the algorithms were set up. By using a similar structure for the functions, they could be easily interchanged. We also isolated common code between parallel sorting algorithms to prevent duplicate code.

Now that we could test our library, it was quickly apparent that there were some major issues with the code already present. Sample sort did not actually conform to any known specification. It instead opted to not sort the local array while still choosing samples equidistantly. Since the code was well structured, rewriting it to support regular and random sampling was easy. Curious to see how this "new" variation, we decided to call mixed sampling, would perform we kept it in. Later on, we also ran into issues consisting of integer overflows as we were handling such large numbers.

Having a feature complete version ready and usable now, the next step was switching from `C` to `C++11` in anticipation of the experimental hybrid solutions. There was little that had to be done as we would stay on the `C` subset of `C++` because of performance and portability concerns. Besides the thread support, the only other thing of consideration was template usage for generic versions of the algorithms. Due to time issues, this feature was ultimately deemed unnecessary as we already decided to test only on unsigned 64-bit integers. Since `malloc` in `C` and `C++` function slightly different we did decide to replace the calls to `malloc` and `free` with the `C++` equivalents `new` and `delete`.

Now that the implementation was finished, we created proper documentation for the relevant library functions using Doxygen [22].

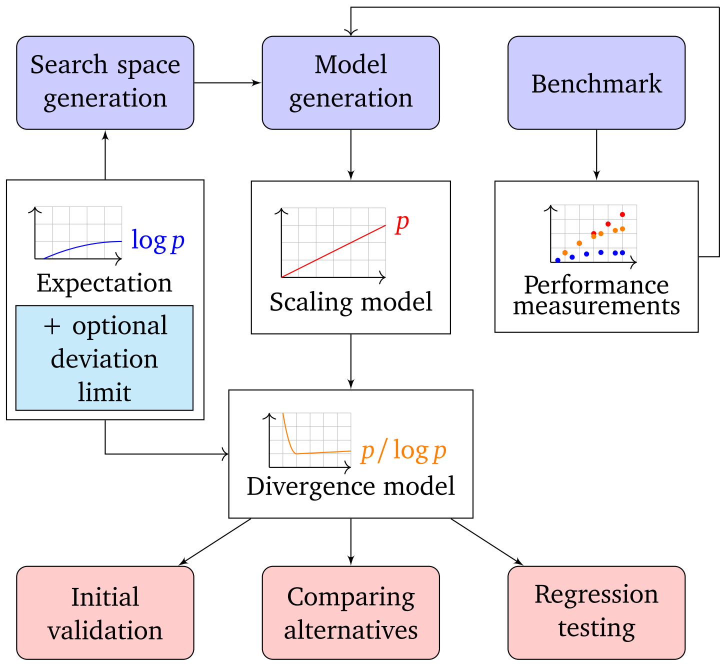


Figure 4.1: Framework overview including use cases. Taken from Exascaling Your Library: Will Your Implementation Meet Your Expectations? [14]

4.2 Validation Framework

The process of validation is based on the scalability validation framework seen in Figure 4.1. It consists of the following four steps:

- Initial expectations
- Benchmark design
- Generating performance models
- Validating expectations

Initial expectations

Thanks to extensive literature about parallel sorting algorithms, we can easily find the initial expectations and incidentally, the validation framework expects the initial expectations in big O notation (Landau notation). In almost all cases the complexity was constructed via analytical performance modeling. For the scalability validation, we also want to further simplify the expression into so-called leading-order terms.

Design benchmark

To create our benchmark data we will use Score-P as we can easily use the data in Extra-P. Initially, there was a significant performance loss and after the change to C++ there were unnecessary and cluttering calls to C++ specific code visible in the call tree. To compensate, but not entirely nullify the performance regression, a filter was utilized. The code is compiled once and then run on an increasing amount of processes via a batch script.

Generate scaling models

For the performance modeling, a developer version of Extra-P is used that implements "Automated Refinement of Performance Models" by Reisert [20]. We also made additional modification to the source code. First, we used the median option opposed to the mean option to generate the performance model. Another problem arose in the occasions when only one process would do the majority of the work. Normally an average across all processes was used which introduced falsely models. We changed the relevant code to always use the maximum value over all processes. A weak scaling approach was chosen to generate the models.

Validate expectations

As we accept expectations in big O notation, we need to adjust the generated performance model. We will drop coefficients that were introduced and simplify the model to its leading-order term. We also introduce a divergence model defined as $\delta(p) = F(x)/E(x)$ where $F(x)$ is the modified performance model expression and $E(x)$ is the expectation. This will help in the evaluation step, as we can see in what capacity the model diverges from the expectation.

4.3 Experimental Hybrid Solutions

Lastly, we will focus on the splitter-based sorting algorithms once more. While we largely base all our performance on the performance of MPI, the splitter-based approaches call for sorting its local array of size n_i. As n_i reaches into the tens of millions of elements and as a result takes up most of the run time of these algorithms, it will oftentimes dominate the performance model of the whole algorithm.

By using only a subset of available processes for MPI, we are able to use the other processes to speed up the local sorting step. We will do this by implementing a simple parallel merge sort for C++11 threads and OpenMP tasks. Assuming a constant n, means each MPI process will have a larger array to sort. The goal of this experiment is to evaluate the loss of performance of cutting down on MPI processes versus the potential speedup of parallelizing the local sorting step.

The underlying sorting algorithms will work on shared memory and thus can make away with the constraints of the earlier parallel sorting algorithms. The downside is that we can not multithread our algorithm between nodes. But thankfully that is not relevant for our use case, as we only need to work with the data of one MPI process. At large, these algorithms will simply try to divide the local array between the available parallel computing units. By multithreading and accelerating the local sorting step, we hope to gain some more insight into how not only the algorithms themselves scale but also how the MPI processes scale with increasing data.

Algorithm 8 Parallel Merge Sort

```
Input list $A$ with $|A| = n$, a lower bound $lo$ and a higher bound $hi$
1: if $lo + 1 < hi$ then
2:     mid $= [(lo + hi)/2]$
3:     fork mergesort($A, lo, mid$)
4:     mergesort($A, mid, high$)
5:     join
6:     merge($A, lo, mid, hi$)
7: end if
8: return $A$
```

A simple implementation of parallel merge sort that we are using is presented in Algorithm 8. Note that the recursion termination of our implementation will also terminate if the size of the partition falls under 32 thousand elements. Furthermore, the multithreaded algorithm will also base how many threads or tasks it will spawn on the remaining available resources. If either the remaining amount of elements is too low or we run out of available cores, a sequential merge sort is used.

For the CUDA accelerated local sorting we will use the sort function provided by CUDA's thrust library [23]. Internally this function has implementations for both parallel radix sort and parallel merge sort. Since radix sort is faster in general, it will be used by default. Whenever the input data is not integer-like and thus not applicable to radix sort, the merge sort implementation is used instead. Before sorting we will also have to copy the data array from the main memory to the GPU memory and back. While this is a trivial concept, it could have big performance implications.

For the performance modeling we will once again use Score-P and Extra-P, but now to test only the splitter-based algorithms. We will apply the validation framework once more but this time we keep p constant and increase n instead. This way n_i is steadily increasing across all processes. Whereas these values are not nicely divisible by p the resulting loss of elements on each process is negligible.

5 Evaluation

In this chapter, we will apply the scalability validation framework to our library 'libparsort'. We will create performance models of the parallel sorting algorithms, analyze them and compare them to the expectations. Additionally, we will discuss the performance of the parallel sorting algorithms against each other. Lastly, we will evaluate the results of our experimentation in parallelizing the local sort of splitter-based algorithms.

5.1 Experimentation Environment

The data we will present was collected by benchmarking our parallel sorting library on the Lichtenberg High Performance Computer [1] at Technische Universität Darmstadt. Using Slurm [25], the job scheduling system used on the cluster, we were able to effortlessly submit jobs that run all the benchmarks at high repetitions.

In Figure 5.1 we can see that the topology of the HPC is split into two phases. For the validation process, we will only use phase I. More specifically we will use one MPI island(i01-i15), which consists of 32 nodes. Each node has 32 gigabytes of main memory and two "Intel® Xeon® E5-2670" processors resulting in 16 cores. Tests will be run from one node with 16 MPI processes to all 32 nodes with 512 MPI processes. Since we are interested in the weak scaling behaviour of these parallel sorting algorithms, we decided to keep the size of the data on each process n_i constant to ten million. Ten repetitions were used for this step.

For the benchmarking of the local sorting experiments, we were limited by CUDA. Figure 5.1 shows an island marked as NVD. It consists of 44 nodes identical to the MPI island nodes but also includes two "NVIDIA® Tesla™ K20X" per node. Assigning a MPI process to each GPU, we can only run two MPI processes per node. As a result, we decided to limit the benchmarking for the local sorting experiment to 64 MPI processes, which correspond to 32 nodes on the NVD

[1] http://www.hhlr.tu-darmstadt.de/hhlr/

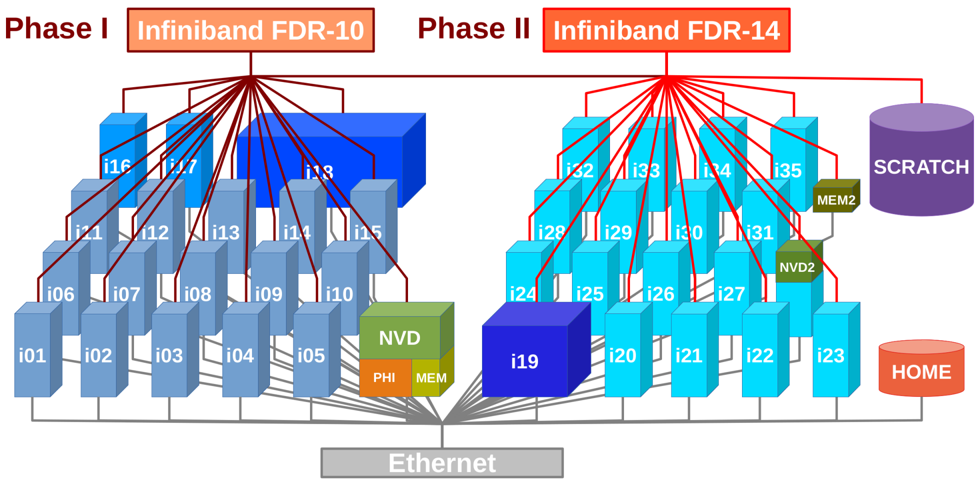


Figure 5.1: Island topology of the Lichtenberg High Performance Computer at Technische Universität Darmstadt. [24] (accessed 2017-10-13)

island. In this case, increasing the amount of processes over the benchmarking process does not make sense as the local sorting step is only affected by n. Instead, we will increase n, which will also increase n_i, from 30 to 240 million elements in eight steps. Five repetitions were used for this step.

A new seed is chosen for every iteration. For changes in the configuration such as increasing n or p, the same seed is kept. This was done to ensure some overlap in the input data between different process configurations. Histogram sort was benchmarked using a threshold q of 10%. Radix sort was run with $k = 8$.

5.2 Validating Expectations

As outlined in the scalability validation framework we do not want to compare the resulting performance models directly to the expectations. We will instead present the expectations in big O notation (Landau notation) and compare only the leading-order term, as it describes the scalability behaviour on highly scalable systems the best. The models Extra-P constructs automatically are unfit for direct comparison because they are constructed in a way where inner functions are weighted based on their run time. That usually results in functions that take more time to mask the leading-order term. Extra-P also adds constant expressions to the model that are largely irrelevant in the context of parallel scalability. That is why the created model and the extracted leading-order term might not have any direct correlation, as the leading-order term might have been taken from an inner function that was not considered for the final generated model.

Table 5.1 summarizes our findings. In the following passages, we will establish the expectations for each algorithm, simplify them to a leading-order term and discuss the results.

Sample Sort

The expectations for splitter-based parallel sorting algorithms are similar in structure. Most notably they will reflect the local sorting step in $\mathcal{O}(\frac{n}{p}\log n)$. With the way we do our benchmarking the expression $\frac{n}{p}$ corresponds to the constant n_i. Furthermore, the expression $\log n$ can be rewritten as $\log p \cdot n_i$ and simplified to $\log p$. That means the local sorting step is expressed as $\mathcal{O}(\log p)$.

Initial expectations:	$\mathcal{O}(\frac{n}{p}\log n + p^2 \log p^2)$
Expected scalability behaviour:	$\mathcal{O}(p^2 \log p)$

For both regular and random sample sort, the leading order term can be traced back to calculating the splitters and distributing them to all processes. With a big O notation of $\mathcal{O}(p^2 \log p)$ it matches our generated model perfectly. While $\bar{R}^2$ is significantly lower for random and mixed sample sort, SMAPE shows that the error is rather small.

Histogram Sort

Histogram is data dependent as shown in the complexity expectation by r, the number of repetitions until sufficient splitters are found. Yet our tests show that given our large local sizes with randomly generated input, the number of repetitions never reached a critical mass where it would affect the scalability behaviour of this algorithm.

Initial expectations:	$\mathcal{O}(r * p \log \frac{n}{p} + \frac{n}{p}\log n)$
Expected scalability behaviour:	$\mathcal{O}(p)$

Table 5.1: Generated runtime models of various parallel sorting algorithms implemented in our *libparsort* library. $\bar{R}^2$ specifies the adjusted coefficient of determination and SMAPE is a scale-independent, relative error metric. $\delta(p)$ represents how far the model diverges from the expectation. The Match column shows if the validation was successful. ⚠ will signal inconclusive behaviour

	Expectation						
Algorithm	Time complexity	Leading-order term	Model leading-order term	$\bar{R}^2$	SMAPE	$\delta(p)$	Match
Sample Sort Regular	$\mathcal{O}(\frac{n}{p}\log n + p^2 \log p^2)$	$\mathcal{O}(p^2 \log p)$	$\mathcal{O}(p^2 \log p)$	1	0.592	$\mathcal{O}(1)$	✓
Sample Sort Random	$\mathcal{O}(\frac{n}{p}\log n + p^2 \log p^2)$	$\mathcal{O}(p^2 \log p)$	$\mathcal{O}(p^2 \log p)$	0.7	1.18	$\mathcal{O}(1)$	✓
Sample Sort Mixed	$\mathcal{O}(\frac{n}{p}\log n + p^2 \log p^2)$	$\mathcal{O}(p^2 \log p)$	$\mathcal{O}(p^2 \log p)$	0.74	2.884	$\mathcal{O}(1)$	✓
Histogram Sort	$\mathcal{O}(r \cdot p \log \frac{n}{p} + \frac{n}{p}\log n)$	$\mathcal{O}(p)$	$\mathcal{O}(p^{2.5} \log p)$	0.967	5.478	$\mathcal{O}(p^{1.5} \log p)$	**x**
Exact Splitting Sort	$\mathcal{O}(\frac{n}{p}\log n + p \log^2 n)$	$\mathcal{O}(p \log^2 p)$	$\mathcal{O}(p^{1.5} \log p)$	0.995	1.844	$\mathcal{O}(p^{0.5} / \log p)$	$\approx$
Radix Sort	$\mathcal{O}(\frac{b}{k}(\frac{n}{p} + 2^k + \log p))$	$\mathcal{O}(\log p)$	$\mathcal{O}(p^{0.5} \log p)$	0.984	2.341	$\mathcal{O}(p^{0.5})$	$\approx$
Mini Sort Sequential	$\mathcal{O}(p \log p)$	$\mathcal{O}(p \log p)$	$\mathcal{O}(p^{0.5})$	0.915	6.816	$\mathcal{O}(1/(p^{0.5} \log p))$	⚠
Mini Sort Counting	$\mathcal{O}(p)$	$\mathcal{O}(p)$	$\mathcal{O}(p^{0.5})$	0.745	76.734	$\mathcal{O}(1/p^{0.5})$	⚠
Mini Sort Scalable	$\mathcal{O}(\log^2 p)$	$\mathcal{O}(\log^2 p)$	$\mathcal{O}(p \log p)$	0.998	13.139	$\mathcal{O}(p / \log p)$	$\approx$ ⚠

Yet the generated model does not match the expectations at all. Even assuming r did actually reach p, the expectation and the model would still not line up. Thus, it is very likely that there is some scalability bug in our implementation.

Exact Splitting Sort

The complexity of exact splitting sort is dominated by the local sorting step and the global splitter finding. As we have established, in this test case the local sorting step should be constant. Thus only the global splitter finding complexity is observed.

Initial expectations:	$\mathcal{O}(\frac{n}{p}\log n + p\log^2 n)$
Expected scalability behaviour:	$\mathcal{O}(p\log^2 p)$

The generated model is not unreasonably different to the expectations. But it still misses the mark and deserves some discussion. The implementation of exact splitting sort includes a set of configurations that include different median algorithms and a custom `alltoallv`. It is possible that either of these potions will have an impacts on the performance of the algorithm. Furthermore, the algorithm was more likely to fail the validation because of its relatively young age. This is one of the first works that incorporate exact splitting sort.

Radix Sort

As radix sort needs to do repetitions based on the number of bits b in the elements and k, the number of bits in a digit, its complexity is largely based on these parameters. With a relatively low number of processes, the expression 2^k will dominate the scalability behaviour. Regardless, the generated model will need to be compared to the leading-order term which is $\mathcal{O}(\log p)$.

Initial expectations:	$\mathcal{O}(\frac{b}{k}(\frac{n}{p} + 2^k + \log p))$
Expected scalability behaviour:	$\mathcal{O}(\log p)$

The difference in expectation and model can be easily explained with the aforementioned phenomenon on low process counts. Our tests only use up to 512 processes where 2^k will still dominate. Further testing is necessary for a conclusive answer, but given the circumstances, the $\mathcal{O}(p^{0.5})$ difference can be attributed to the aforementioned problem and additional noise.

Mini Sort

Due to the simple nature of mini sort, we do not need to extract the leading-order term from their run time complexities. You can see the expectations in Table 5.1.

Interestingly enough almost all of the variations perform better in the performance model than in the expectations. This result can be traced back to the inadequate testing environment for an algorithm such as mini sort. Part of that can be seen in the SMAPE metric as well as $\bar{R}^2$. Further testing on a more scalable environment is needed.

5.3 Performance Comparison

Another point of interest is how the algorithms perform against each other. Given the expectations, we can already guess, which will perform the best. And considering the validation process, we can confirm our guess. We plot the generated models onto the same coordinate system that we can see in Figure 5.2.

Mini sort was not included as the experimentation parameters differ to heavily from the other parallel sorting algorithms. As for sample sort, only regular and random sample sort were observed. We can see that by using the generated models instead of the abstracted asymptotic

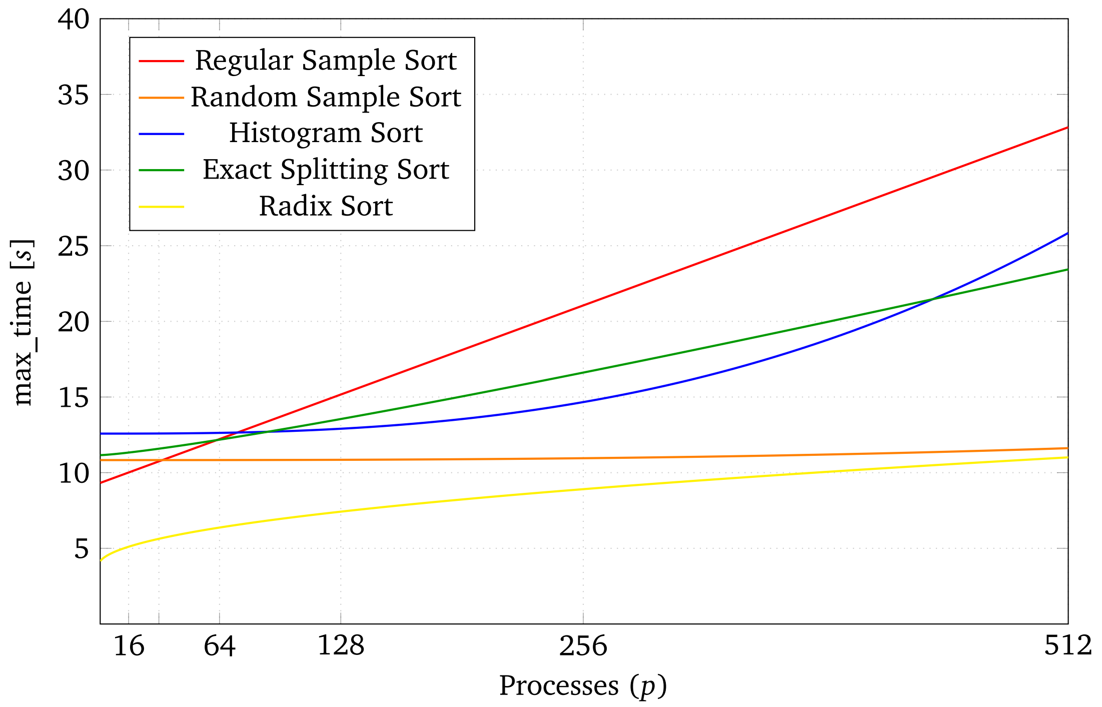


Figure 5.2: Performance comparison of the parallel sorting algorithms.

scalability behaviour, some of the algorithms are not accurately represented. Most notably regular and random sample sort demonstrate far better scalability performance than they should. This clearly showcases how inconclusive the data is. Nonetheless, by cross referencing with Table 5.1, we can safely assume that radix sort will perform the best on our input.

5.4 Parallel Local Sort

Here we make use of another aspect of the scalability validation framework. Instead of comparing our findings to expectations from literature, we run our benchmark under previous conditions and create a performance model first. This will present our initial model from which we can conclude the general scalability of our algorithms. Afterwards, we can make changes to our code, more specifically to our function that handles the sorting of the local data of each process. We will compare the generated models directly because the expected scalability behaviour is unlikely to show in the model and because we want to see the speed up that was achieved. We decided to only observe regular sampling for sample sort for convenience.

5.4.1 C++11 Threads/OpenMP

OpenMP tasks and C++11 threads behave quite similarly. But their performance is dependant on the available resources. As outlined in Section 5.1, we will keep the number of processes constant but we will run separate benchmarks on different amount of MPI processes. We will start with 32 MPI processes over 4 nodes and go down to 4 MPI processes, which means 1 MPI process per node. The remaining cores will be used to run the threads or tasks. For example, with 32 MPI processes, each MPI process will be able to use two threads/tasks.

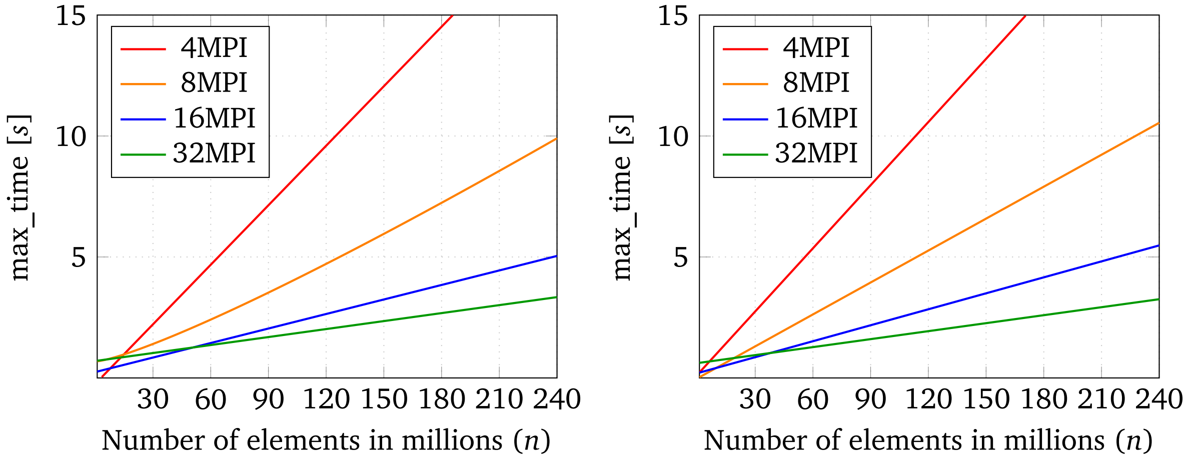
In Figure 5.3 we can see that the performance worsens with the increase in threads/tasks. This might be because of our very simple approach to the parallel merge sort that has very limited scalability. Besides an outlier seen in Figure 5.3b the data makes sense given that the amount of data that is sent between processes doubles when we half the MPI processes used.

Comparing the best performing multithreading approach on 32 MPI processes to the sequential merge sort used on 64 MPI processes we can see an improvement in performance on our input parameters. Given the scalability trend, the sequential merge sort will eventually still perform better on larger input sizes. This can already be seen in Figure 5.4 for exact splitting sort.

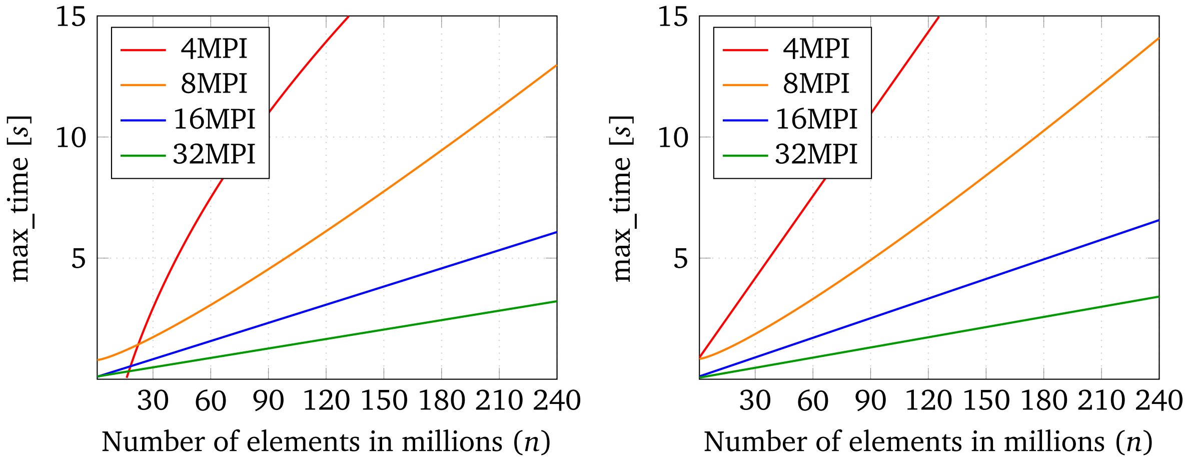
5.4.2 CUDA

Testing the CUDA implementation will be much simpler since we only have one configuration. Each node will have two MPI processes that each get to use one of the two graphics cards for accelerated sorting.

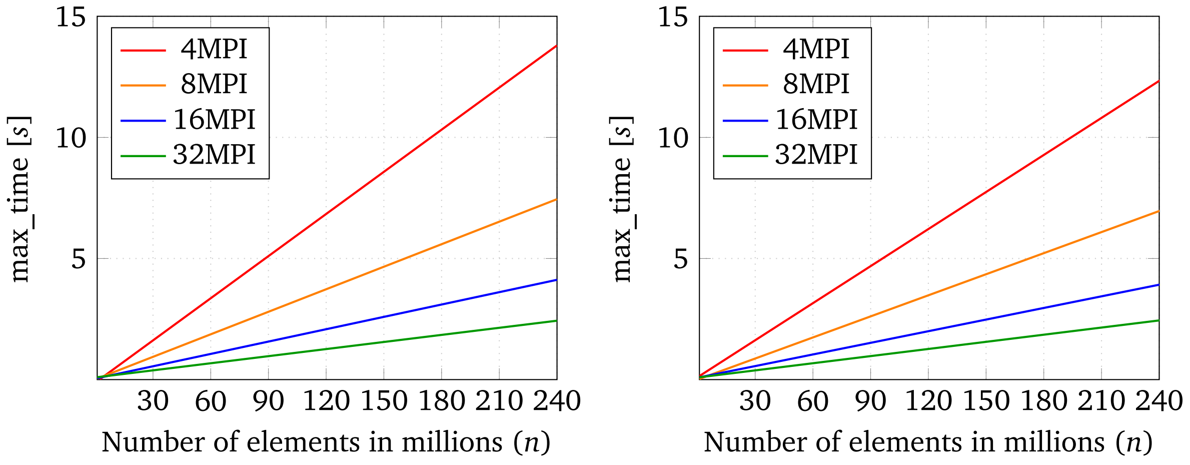
The results in comparison with the multithreaded solution and the sequential merge sort can be seen in Figure 5.4. While it looks promising, the accuracy of the data needs to be questioned. Given that the data needs to be copied to the GPUs memory, this outcome is more than likely inconclusive. Compared to exact splitting sort where the model for CUDA was too big to be modeled it is safe to assume that the model behaves very irregularly.



(a) Regular Sample Sort

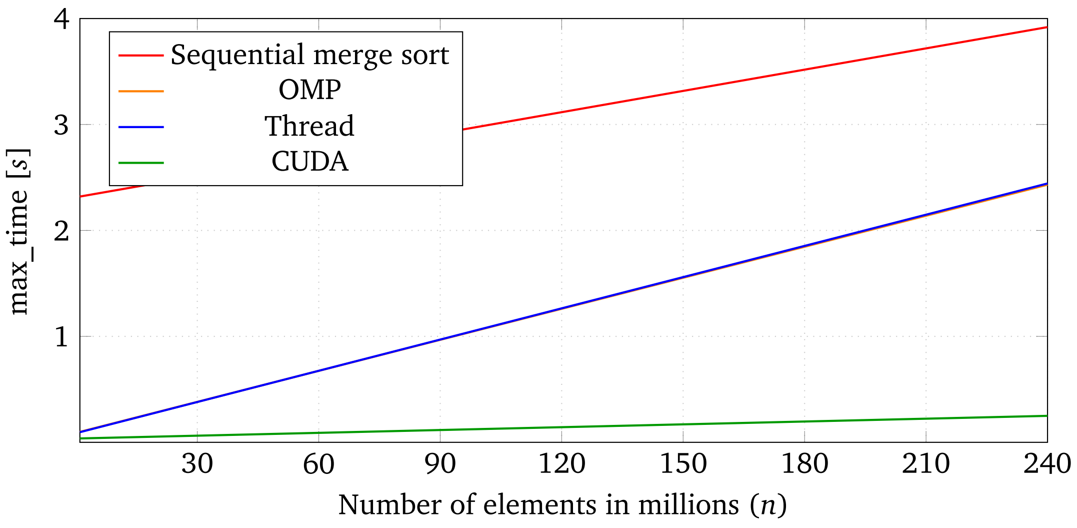


(b) Histogram Sort

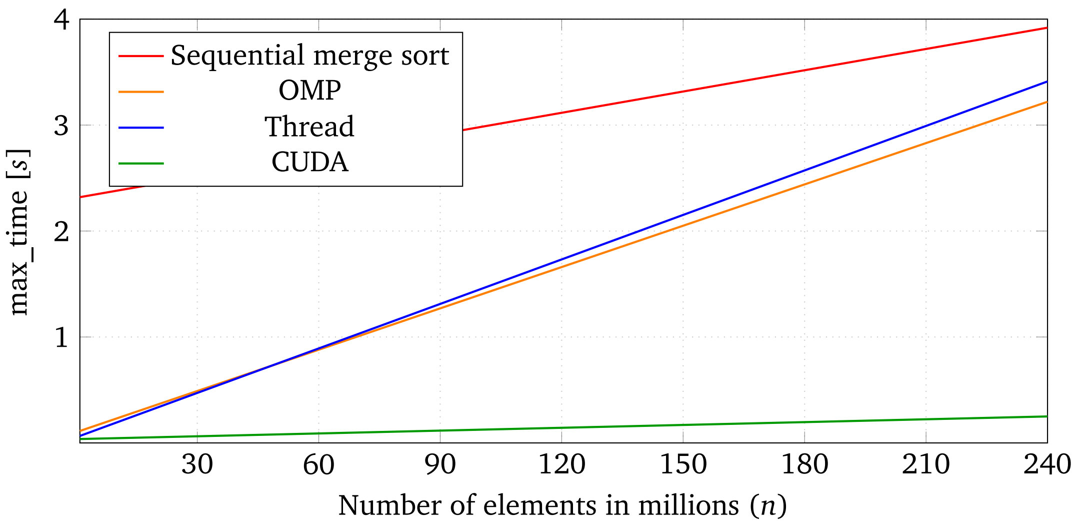


(c) Exact Splitting Sort

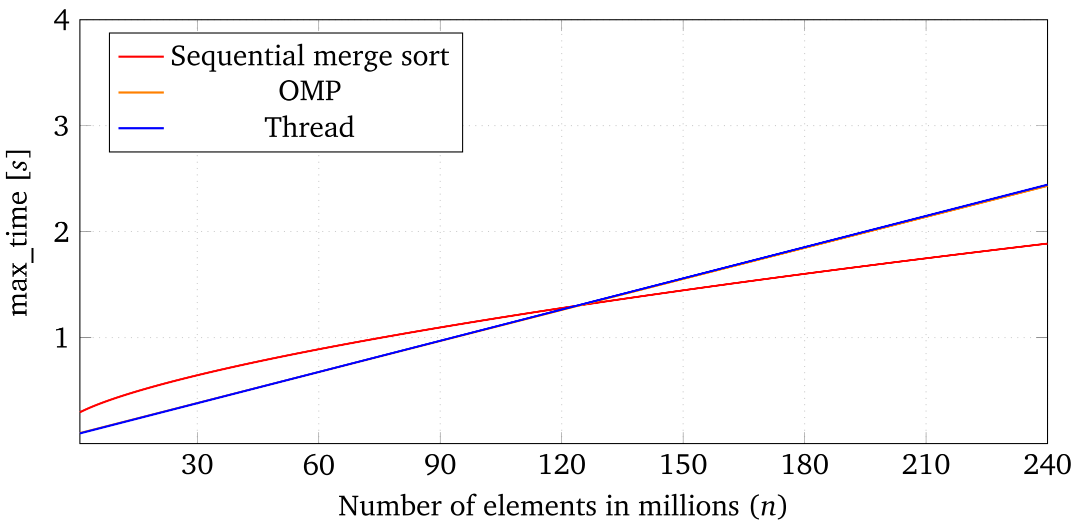
Figure 5.3: Comparison of parallelization with OpenMP tasks(left) and C++11 threads(right) on different amount of MPI processes.



(a) Sample Sort



(b) Histogram Sort



(c) Exact Splitting Sort

Figure 5.4: Comparison of all forms of parallelization of the local sorting step using two C++11 threads / two OpenMP tasks per MPI process.

6 Conclusion & Outlook

Empirical performance modeling presents a way to generate run time models from benchmarks. This can be extended to represent the scalability behaviour of a given application on increasing number of processes. Traditionally, analytical performance modeling was used to express the run time behaviour. But with highly complex and parallel code, there is no proof that these expectations are accurate. But unlike previous work, we do not construct the run time complexities analytically but with empirical performance modeling using Extra-P.

In this thesis, we discussed the idea of parallel computing and parallel sorting. As part of this discussion, we presented five state-of-the-art parallel sorting algorithms. The splitter-based algorithms sample sort, histogram sort, and exact splitting sort. As well as parallel radix sort and mini sort.

Consolidating these algorithms into a coherent library, we were able to generate empirical performance models using Score-P and Extra-P. Afterwards, we were able to use the models in a scalability validation framework to validate the expectations and explore possible improvements by parallelizing the local sorting of the splitter-based parallel sorting algorithms. While the results for the experimental parallelization are largely inconclusive, it shows promise and should be more closely investigated. We are however able to show that the expected scalability behaviour of the parallel sorting algorithms matches in for a large part of the algorithms.

But there is a point to be made that different experimentation environments might present data that will lead to another conclusion. Especially for mini sort, as its innate design requires a very large number of processes. The scalability behaviour over different architectures might be a basis for future work.

Furthermore, we already presented some possible improvements that can be made to the core of the parallel sorting library. These improvements target the specification of the algorithms in histogram sort's case or suggest the incorporation of implementation specific improvements. Exact splitting sorts p-way-merge algorithm and its custom `alltoallv` MPI collective function might help to improve sample sort or histogram sort.

Future works could also attempt to classify the parallel sorting algorithms. Classifications would include the performance model based on a two-parameter approach over p and n and architectural identifiers such as network topology and nature of the input data. This might make it possible to automatically choose the parallel sorting algorithm that best fits the use case.

Bibliography

[1] W. D. Frazer and A. C. McKellar. Samplesort: a sampling approach to minimal storage tree sorting. *J. ACM*, 17(3):496–507, July 1970. ISSN: 0004-5411. DOI: `10.1145/321592.321600`.

[2] L. V. Kale and S. Krishnan. A comparison based parallel sorting algorithm. In *Parallel Processing, 1993. ICPP 1993. International Conference on*, volume 3, pages 196–200, Aug. 1993. DOI: `10.1109/ICPP.1993.17`.

[3] C. Siebert and F. G. E. Wolf. A scalable parallel sorting algorithm using exact splitting. Technical report, Aachen, 2011, 10 S.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. Part Chapter 9. Sorting. ISBN: 0262033844.

[5] C. Siebert and F. Wolf. Parallel sorting with minimal data. In Y. Cotronis, A. Danalis, D. Nikolopoulos, and J. Dongarra, editors, *Proc. of the 18th European MPI Users' Group Meeting (EuroMPI), Santorini, Greece*, volume 6960 of *Lecture Notes in Computer Science*, pages 170–177. Springer, Sept. 2011. ISBN: 978-3-642-24448-3. DOI: `10.1007/978-3-642-24449-0_20`.

[6] Extra-P. URL: `http://www.scalasca.org/software/extra-p/download.html` (accessed 2017-10-13).

[7] Score-P. URL: `http://www.vi-hps.org/projects/score-p/` (accessed 2017-10-13).

[8] C++ Threads. URL: `http://en.cppreference.com/w/cpp/thread/thread` (accessed 2017-10-13).

[9] OpenMP. URL: `http://www.openmp.org/` (accessed 2017-10-13).

[10] CUDA. URL: `https://developer.nvidia.com/cuda-zone` (accessed 2017-10-13).

[11] C. Siebert, E. Peise, and F. Wolf. Suitability of Parallel Sorting Algorithms for Dynamic Load Balancing. unpublished,

[12] G. E. Blelloch, C. Leiserson, B. M. Maggs, C. G. Plaxton, S. J. Smith, and M. Zagha. An experimental analysis of parallel sorting algorithms. 31:135–167, Apr. 1998.

[13] N. Amato, R. Iyer, S. Sundaresan, and Y. Wu. A Comparison of Parallel Sorting Algorithms on Different Architectures. Technical report, College Station, TX, USA, 1998.

[14] S. Shudler, A. Calotoiu, T. Hoefler, A. Strube, and F. Wolf. Exascaling your library: will your implementation meet your expectations? In *Proc. of the International Conference on Supercomputing (ICS), Newport Beach, CA, USA*, pages 165–175. ACM, June 2015. DOI: `10.1145/2751205.2751216`.

[15] S. White, N. Verosky, and T. Newhall. A CUDA-MPI Hybrid Bitonic Sorting Algorithm for GPU Clusters. Technical report, Sept. 2012, pages 588–589.

[16] L. Shenghui, M. Junfeng, and C. Nan. Internal sorting algorithm for large-scale data based on GPU-assisted. Technical report, Aug. 2013, pages 634–638.

[17] A. Radenski. Shared memory, message passing, and hybrid merge sorts for standalone and clustered smps,

[18] E. Solomonik and L. V. Kalé. Highly scalable parallel sorting. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, Apr. 2010. DOI: 10.1109/IPDPS.2010.5470406.

[19] Cube 4 – Performance report explorer and data format. URL: http://www.scalasca.org/software/cube-4.x (accessed 2017-10-13).

[20] K. P. Reisert. *Automated Refinement of Performance Models*. Master Thesis, Department of Computer Science, Laboratory for Parallel Programming, Apr. 2017.

[21] Open MPI v1.10.7 documentation. URL: http://www.open-mpi.de/doc/v1.10/ (accessed 2017-10-13).

[22] Doxygen. URL: http://www.stack.nl/~dimitri/doxygen/ (accessed 2017-10-13).

[23] Thrust. URL: https://thrust.github.io/ (accessed 2017-10-13).

[24] Lichtenberg High Performance Computer of TU Darmstadt. URL: http://www.hhlr.tu-darmstadt.de/hhlr/index.en.jsp (accessed 2017-10-13).

[25] Slurm Workload Manager – Documentation. URL: https://slurm.schedmd.com/ (accessed 2017-10-13).