Uppsala University

High Performance and Parallel Computing

1TD065 62035

Project
Parallel Quicksort

Tobias Lass

June 2, 2023

Contents

1 Introduction

Sorting sets of elements has been a vital part of computation since its conception and continues to be at the core of modern computation. While generally optimal algorithms have been around for decades, new perspectives appear as our needs as a society and engineers change [1]. With the limitations of single processor improvements approaching, the need for efficient multi-processor computation is ever increasing [2]. This project explores a parallelization of the *Quicksort* algorithm using the *C programming language* and the *pthreads* library.

2 Problem

A set of orderable elements $A = \{a_1, ..a_i, .., a_n\}$ can be arranged into an ordered list $B = [a_{j,1}, .., a_{j,i}, .., a_{j,n}]$ that has the sorted property exactly when $a_{j,i} \leq a_{j,i+1} \ \forall \ 1 \leq i < n$. The problem is to efficiently and in parallel take an input of randomized elements and produce an output of an ordered list of those elements.

The classic *Quicksort* algorithm employs a divide and conquer strategy that recursively partitions a set of elements into a lower and an upper part according to a chosen pivot element and eventually achieves the sorted property once the set only contains one element [3]. However, the algorithm described in the project specification is more closely related to a *Merge-Splitting Sort* [4]. A strict parallelization of quicksort would intuitively be implemented with task-based parallelism - creating new, independent tasks with each recursion until a specific limit is reached and then completed locally.

The adapted approach of merging and splitting local subarrays introduces the difficulty of inter-process communication in the form of exchanging data and synchronization. This algorithm may be conveniently implemented using the *OpenMPI* library with its native support of splitting process communication into groups [5] and the recursive nature of this algorithm. However, this project focuses on shared memory parallelism using either *OpenMP* or *Pthreads*. *Pthreads* was chosen for its more low-level control over threads [6].

3 Solution

3.1 Algorithms

The algorithm is illustrated as pseudocode in listing 1. First, an input sequence xs is split up over T threads and sorted locally in the preprocessing step. Then, each processor is assigned to a group of size t - initially $t = T$. Next, a pivot element p is chosen within a group and all threads find a split point s such that all elements $\leq p$ are located below s and all elements $> p$ are located on s or above in their local subarray. According to the position of the thread within its group, it communicates its corresponding split with a partner thread in the opposite part of the group. I.e., a thread in the lower half of a group communicates its upper split to a thread in the upper half of a group and vice versa. Each thread then merges its local and remote split on a new local subarray. As the recursive step, the number of threads t is halved. This is repeated until $t = 1$ and the local subarrays can be merged back into a global array.

The local sort in the preprocessing step is implemented with a classic quicksort - choosing the central element as the pivot element. Finding the split point s given a pivot element p is implemented with an inherently failing *Binary Search* after which the index is shifted upwards to the correct position.

The selection of the pivot element per group is offered as 3 different strategies:

1. select the median of the local array of the first thread in each group,
2. select the mean value of all local medians in each group,
3. select the mean value of the center two medians in each group.

Listing 1: Pseudo Code for Parallel Quicksort

```
parallel_qs(xs, T, id):
    ys = scatter(xs, id)
    serial_qs(ys)
    t = T
    while (t > 1):
        lid = id % t
        gid = id / t
        synchronize_group(gid)
        p = select_pivot(gid)
        s = split(ys, p)
        synchronize_partner(lid)
        if (lid < t / 2):
            zs = merge(ys, s, get(lid + t / 2))
        else:
            zs = merge(ys, s, get(lid - t / 2))
        synchronize_partner(lid)
        ys = zs
        t = t / 2
    xs = gather(ys, id)
```

3.2 Data Structures

The basic structure consists of a global array xs, local subarray ys, and temporary local subarray zs for merging - requiring $3N$ space for the data. Additional allocation with, usually, the size of the number of threads T are mainly for inter-process data exchange and synchronization, including but not limited to the sizes of the local subarrays, the local medians, each group's pivot element, an array of pointers and sizes of the exchange, and lastly the group and partner *pthread barriers*. All parameters are gathered into structures that are passed to the thread function. There is no return value as the sorting comes into effect in-place.

While the count for the partner pthread barriers is naturally 2, the count for each group barrier is equal to its number of members. Notably, this algorithm requires T to be a power of 2 such that $T = 2^k$ since the number of threads per group is halved in each step. However, this allows for a static allocation of the group barriers and their counts with some arithmetics.

This implementation specifically sorts a set of C *integer* values where the total number of elements N is also required to be an *integer* value. The number of threads T has to be a valid C *unsigned short*.

3.3 Optimization

The program is generally optimized by being memory- and thread-safe. Data types are kept as minimal as possible, e.g. T and variables that deal with it are defined as *unsigned short* as it is unlikely that it will be larger than 65,535 [7]. Less memory usage improves performance. However, the difference is not measurable for this program so it is more of a conceptual optimization.

Similarly, some arithmetic optimizations were made. Defining, e.g. T, as unsigned makes divisions with them faster. Divisions and multiplications by 2 are implemented as a bit-shift by 1 instead. Other than that there are no more integer arithmetic optimizations and most work is based on comparisons and write operations. The difference is not measurable here either.

All functions are implemented as static and all variables are defined as constant where applicable. As the program is compiled with the `-O3` and `-march=native` flag, it enables `gcc` to automatically inline functions and vectorize some loops. However, since there is no data parallelism here, the benefits of vectorization are negligible here. The `-ffast-math` flag is also included too but the only floating point number in the program is the time measurement so it is only a conceptual optimization.

Algorithmically speaking, the local sort is implemented as classic *Quicksort* which performs very well for general cases. An iterative implementation or reducing the recursion depth with *Bubble Sort* for low sizes did not improve it. Selecting the central element as a pivot resulted in the same performance for an average case but greatly improved the performance for sorted input.

The determination of the split point in each subarray is implemented with *Binary Search* which is most likely to fail, thus why the termination condition is preemptively omitted. Instead, the last upper boundary is picked and shifted up until the end of the array is reached or the first element $> p$. Considering that the median can be assumed to be very similar between local subarrays within the same group, another approach would have been to start in the center and then do a *Linear Search* up or downwards. For most threads, this would result in a non-measurable time but the *Binary Search* approach still yielded the best performance on average. Either way, the difference is barely measurable and again rather conceptual. Another very minor optimization is to arrange the conditionals in the last loop of the `split` function to have the more likely expression to be first in the *or* operation, saving some evaluations on the second expression.

The *string* library is used to move large chunks of memory faster with `memcpy` and is used for scattering and gathering the global array into/from local subarrays. At some point in development, there was the idea to use `realloc` to optimize a `free` plus `malloc` call but it randomly corrupted the memory. The number of traversals over N and memory copies is kept to a minimum. Since each local subarray is sorted, a thread determines the parameters for merging its local and the remote split by shifting the pointer on its subarray and the pointer on its subarray for its partner thread according to its split point and just passing the number of elements starting from that pointer. Related to that, there is some pointer arithmetic, e.g. `xs + i` instead of `&xs[i]`, which saves a bit of time not dereferencing and referencing pointers.

Compiling the program with the `-fopt-info-vec` flag shows that a few loops and code blocks have been vectorized. However, the only meaningful loop vectorization is for the trivial copy loops without comparisons in `merge`. While invariants and branching were kept outside of loops where possible, better vectorization does not seem to be possible as there is no low-level data parallelism.

3.4 Parallelisation

Here, *to synchronize* means having threads wait at a *pthread barrier*.

A big chunk of the work consists of locally sorting chunks of the input array. This presents a kind of data parallelism that each thread can perform independently. The rest of the algorithm however requires synchronization at multiple points. Without synchronization, there would be data races making the program non-deterministic and incorrect.

For pivot strategies 2 and 3 a group needs to be synchronized before the pivot element is picked because otherwise, the local median might not have been updated yet. Similarly, a group needs to be synchronized after picking the pivot element too because theoretically, the local median of a thread may change before another thread reads it. Therefore, the determination of the pivot element is limited by the slowest thread and then might as well only be computed by 1 thread in the group.

Afterward, only thread pairs need to be synchronized. A thread needs to wait until its partner thread has found its split point and entered the corresponding pointer and size into the shared buffers. Then, a thread merges its spit with its partner thread's split on a new buffer. The pair needs to be synchronized after completing the merge again because each thread frees the pointer to its local subarray and swaps the newly allocated pointer for

merging in. Freeing a pointer while the other thread is still traversing the buffer would result in memory corruption.

After all steps are completed, the threads are globally synchronized to coordinate gathering the local subarrays back into the global array. Alternatively, the threads could return their local subarray from their thread function and then let the root process assemble them. However, the additional synchronization with a fully parallel memory copy performs better on average than hoping that thread 0 finishes before thread 1, etc.

The parallelization and synchronization may be implemented more easily using *OpenMP* by just using global barriers. This would seem sufficient because all threads have to be synchronized for the gathering anyway. However, using the fine-grained barriers of *Pthreads* allows threads to release a core while idling at some barrier and then have the kernel schedule a different thread on that core. This is negligible for $T \leq C$ where C is the total number of available threads on all cores but makes a difference for $T > C$.

4 Experiments

All experiments were conducted on *vitsippa.it.uu.se* which has an *AMD Opteron (tm) Processor 6282 SE* CPU with 2 sockets, 8 cores per socket, 2 threads per core, resulting in 32 total available threads. Timings are taken after reading the input file and before freeing the global array. All other resource allocations, thread forking/joining computations, synchronizations, and resource deallocation is included in the timing. All measurements were taken 3 times and selected by the median. It is to be noted that another student was training a machine learning model during most of this project, continuously occupying 40% of the total 3200% CPU usage.

4.1 Correctness

The input was generated separately from the main program. The input format for the program is a binary file containing N amount of integer values. A generator program was written to produce different kinds of input distributions.

1. Random Order: integers from 0 to $N-1$, applied *Fisher-Yates Shuffle* [8]
2. Random Int: random integers
3. Sorted: sorted integers from 0 to $N-1$
4. Reversed: sorted integers from $N-1$ to 0

The sorted property is trivial to test as a single traversal comparing each element with its successor element can prove that. However, this does not account for possibilities in which elements go missing, e.g. the whole array is set to zeroes. For this case, the input file was read again and only sorted using the serial classic *Quicksort*, trusting in its correctness as it has been faithfully implemented from reputable sources.

For input that is generated with contiguous numbers from 0 to $N-1$ both the sorted property and completeness can be checked trivially by checking if $xs[i] = i \ \forall \ 0 \leq i < N$. Alternatives to checking completeness would be e.g. counting the number of sevens before and after sorting the array or manually checking the output. The latter has been done but both are not entirely reliable.

4.2 Results

First, the different pivot selection strategies were tested for various types of distributions as seen in figures 1,2,3,4 with $N = 5 \cdot 10^8$. The size was chosen as it was the maximum file size that could be stored on the department machines' file system. The serial times used as a base for the speedups can be seen in table 1.

Distribution	Strategy	time (s)
Random Order	1	83.921876
	2	83.926470
	3	84.693210
Random Int	1	84.349614
	2	85.686563
	3	84.423212
Sorted	1	35.260347
	2	34.226098
	3	34.221137
Reversed	1	38.501118
	2	38.493650
	3	39.152686

Table 1: Runtimes for $T = 1$ with $N = 5 \cdot 10^8$.

Table 1 shows similar runtimes for random distributions and then for sorted inputs whereas the runtime for sorted inputs is less than half. It should be noted that the pivot selection strategy has no impact on the performance as the part of the code is never reached for $T = 1$. However, it is included here for the sake of completeness for the base value for the speedups.

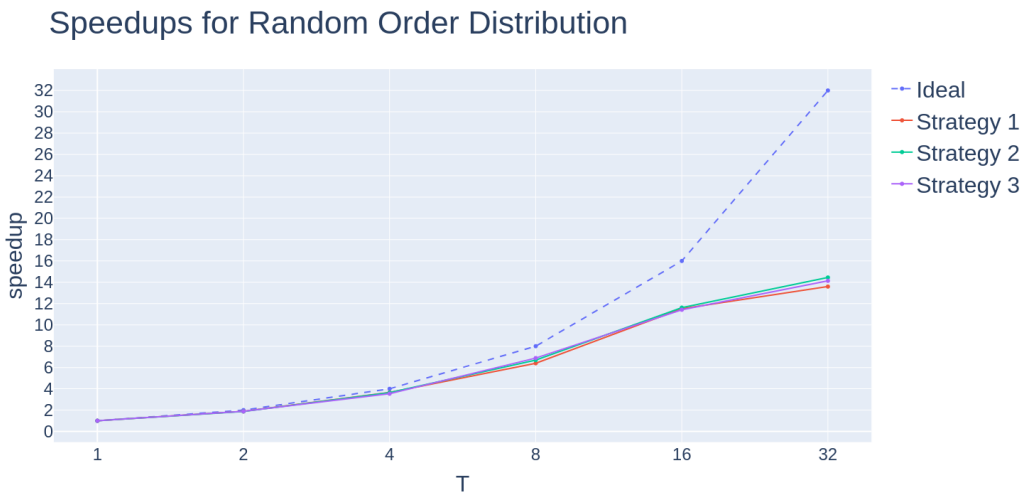

Figure 1: Speedups for Random Order Distribution with $N = 5 \cdot 10^8$.

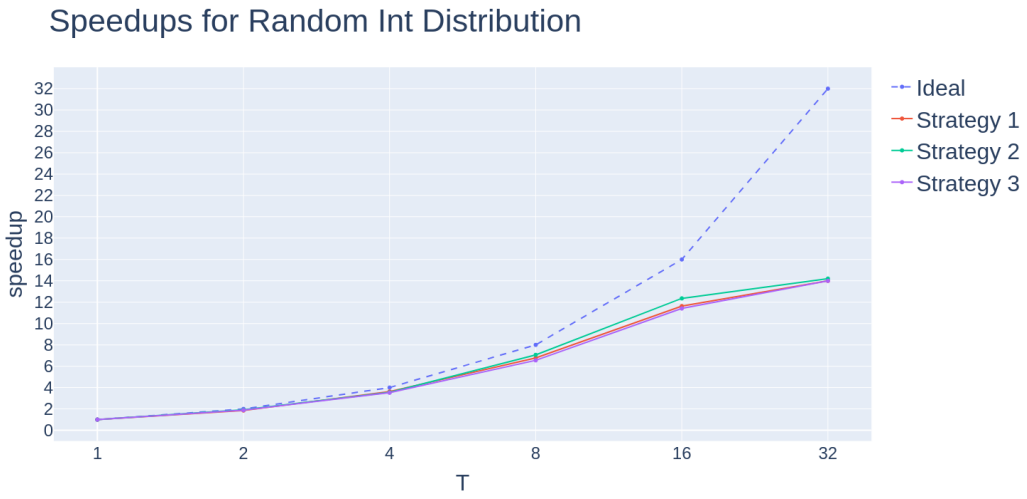

Figure 2: Speedups for Random Int Distribution with $N = 5 \cdot 10^8$.

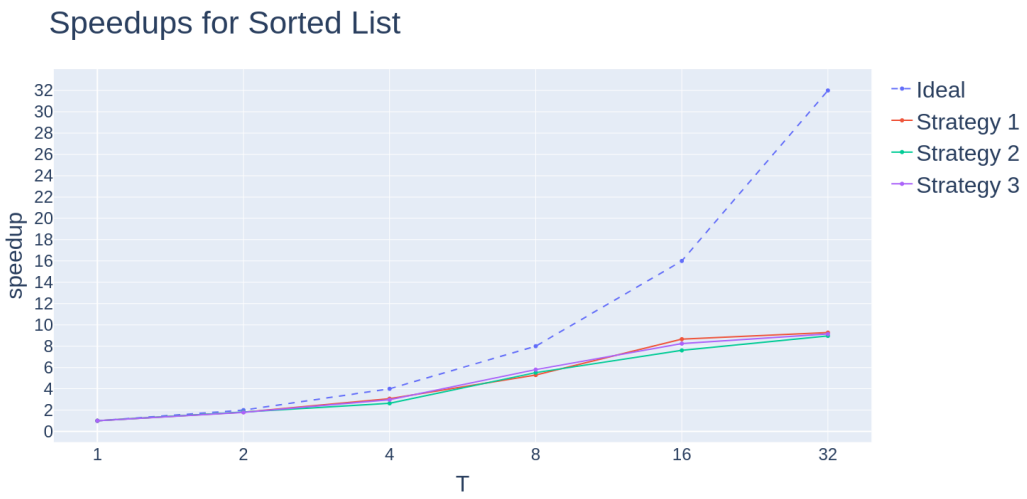

Figure 3: Speedups for Sorted Distribution with $N = 5 \cdot 10^8$.

Figure 4: Speedups for Reversed Distribution with $N = 5 \cdot 10^8$.

Figures 1,2,3,4 show comparable speedups for random distributions and then for the sorted input. The speedups for the different strategies do not appear to diverge much. All variations are most likely explained by arbitrary kernel scheduling and noise on the server. The same goes for the sorted input except that the overall speedup is much lower than for random distributions. The speedup for random distributions stays fairly close to the ideal for $T \leq 16$ with up to 14 and then diverges strongly for $T = 32$.

For subsequent measurements, the Random Int Distribution was picked together with pivot strategy 2 as it seemed to be the most stable and optimal.

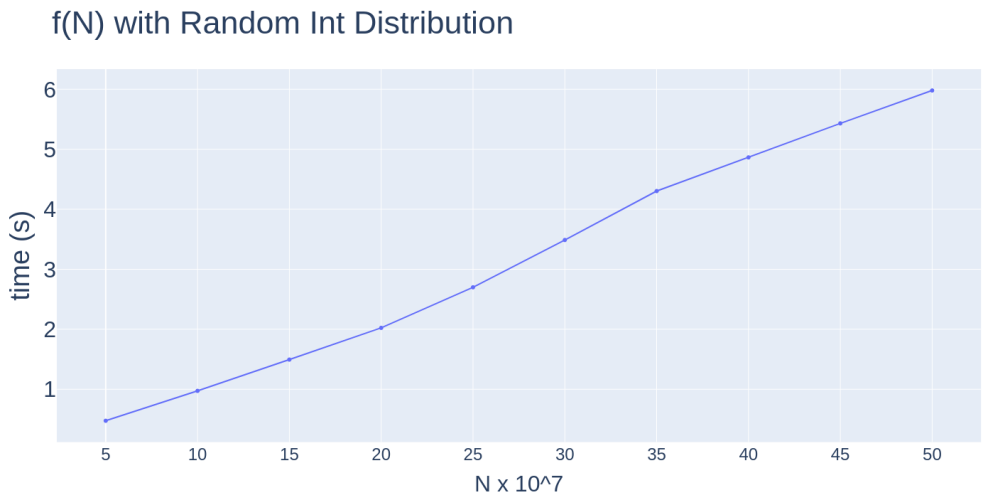

Figure 5: f(N) with $T = 32$ using Random Int Distribution and strategy 2.

Figure 5 shows the runtime of the program as a function of N with $T = 32$. The result is almost linear to the input size.

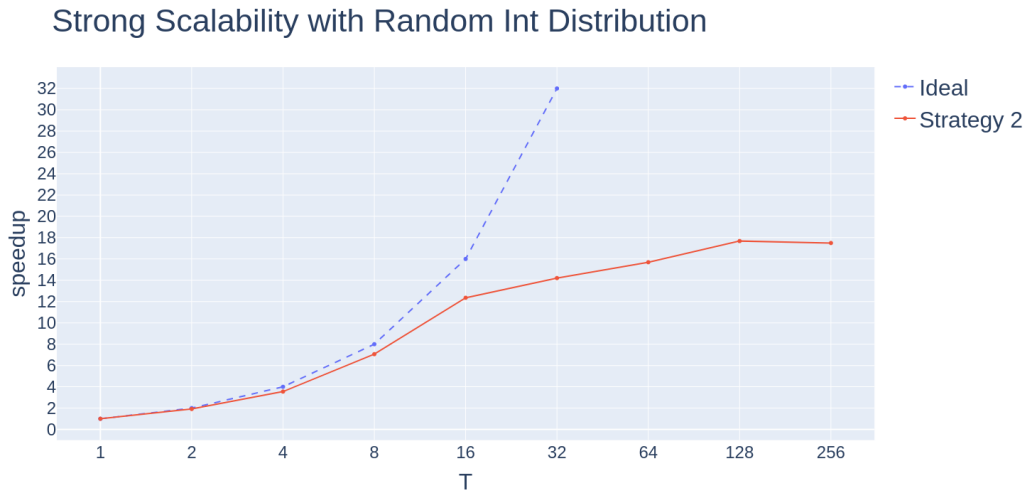

Figure 6: Strong Scalability with $N = 5 \cdot 10^8$ using Random Int Distribution and strategy 2.

Figure 6 highlights the strong scalability of the program also showing the

speedup with more threads than are physically available on the system. The speedup does indeed increase for $T = 64$ and 128. The ideal speedup is not shown for $T > 32$ as it would stretch the y-axis too much. It should be equal to T.

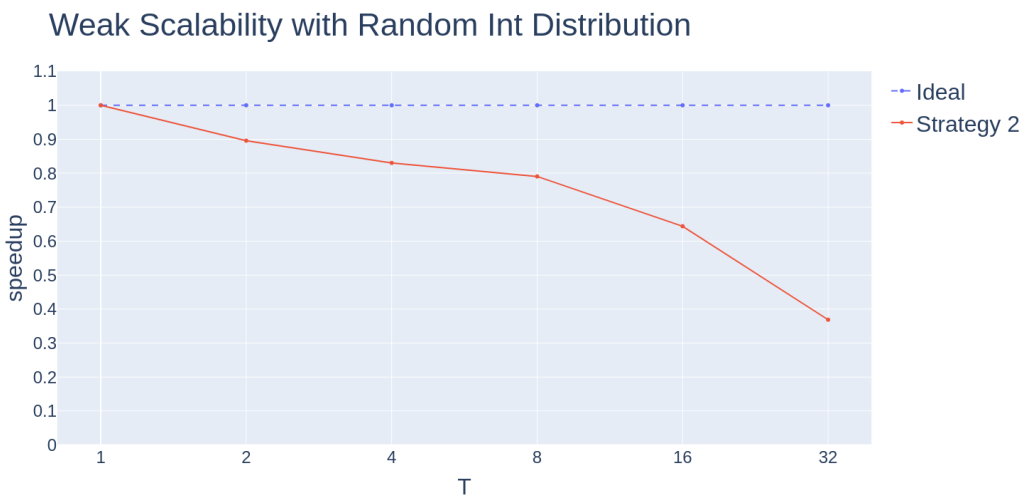

Figure 7: Weak Scalability with $N = T \cdot 10^7$ using Random Int Distribution and strategy 2.

Figure 7 shows the weak scalability of the program for which the workload per thread is kept constant as the number of threads is increased. This means increasing the size of the input by an equal factor to the number of threads, i.e. $N = T \cdot 10^7$. It can be observed that the speedup stays relatively close to the ideal for $T \leq 8$ and then takes a dive.

5 Conclusion

5.1 Discussion

Using *operf* shows that the majority of the time spent is actually in the local sorting step. This explains why the runtimes are so much lower for sorted distributions as seen in table 1. When picking the center element in a sorted array, there are no swapping operations to be done and in a reversed list each element is only swapped once.

This also explains why the speedup is much better for the random distributions because a much larger part of the runtime is spent on perfectly parallel computation. The thread creation overhead and synchronization time carries much more weight for the sorted input. *Amdahl's Law* [9] suggests that the speedup improves when the serial part of the program is reduced which explains the great speedup for random distributions.

The low variation in the performance of the different strategies as seen in figures 1,2,3,4 suggests that test distributions were not perfectly tailored to their strengths and weaknesses. E.g. there was no test distribution for extreme medians which might explain why strategy 2 generally performed slightly better as it made for the most equal load balance on average.

The time measured as a function of N as seen in figure 5 is fairly unremarkable as the time increases almost perfectly linearly with N.

The weak scalability experiment as seen in figure 7 shows a good result meaning the synchronization costs are fairly low for $T \leq 8$ suggesting a good load balance for the chosen distribution and strategy as threads are arriving at almost the same times at barriers.

The strong scalability experiment as seen in figure 6 is the most interesting as it shows a notable increase in speedup for more threads than are physically available up until $T = 128$ which is 4 times as high as the physically available threads. This means a thread may reach a barrier and then release the core while it is idling which allows the core to take on the work of another thread and thus reduce the time an actual core would be idling. This is made possible by fine-graining the synchronization to groups and thread pairs. Since the thread creation and synchronization overhead is fairly significant, it is impressive that the speedup keeps increasing until $T = 128$ and only diminishes for $T = 256$.

5.2 Improvements

Some improvement ideas that were attempted have been discussed in the *Optimization* section.

Checking for correctness could be improved e.g. by counting the values of the unsorted input and the sorted output each into buckets and then comparing them in addition to checking whether the output is sorted. However, this would have required the inclusion or implementation of a *Hashmap* and has thus been rejected in favor of trusting the manually evaluated correctness of the classic *Quicksort* implementation.

To play towards *Amdahl's Law*, the serial part of the program could have been slightly reduced by distributing some of the resource allocation, e.g. for the barriers, to the threads instead of letting the root process do it alone. However, the potential improvement was not deemed high enough to implement this.

As explored in the labs, reducing the number of `malloc` and `free` calls improve the performance. In theory, each thread could keep a buffer of size $N \cdot 2$ for its local subarray and temporary local subarray for merging and keep writing over that memory. However, this would increase the memory usage of the program extremely and was deemed most likely to decrease the performance.

No real cache optimizations have been attempted. The performance of the program is mainly influenced by the local sorting and then merging a part of the local array and another part from a partner thread which cannot sensibly be implemented with blocking. Blocking would probably not increase the performance anyway as for merging the arrays are only traversed once and each memory is only loaded into the cache once anyway.

As for the choice and implementation of the serial sorting algorithm, *Quicksort* was picked as a good, cache-friendly choice for average cases. In contrast, the book on parallel sorting algorithms [4] suggests using *Heapsort* which has a better worst-case performance for the preprocessing step. The choice and specification of the serial sorting algorithm should simply be chosen depending on the characteristics of the input data.

References

[1] C. Bunse, H. Höpfner, S. Roychoudhury, and E. Mansour, "Choosing the" best" sorting algorithm for optimal energy consumption." in *ICSOFT (2)*, 2009, pp. 199–206.

[2] R. Schaller, "Moore's law: past, present and future," *IEEE Spectrum*, vol. 34, no. 6, pp. 52–59, 1997.

[3] C. A. Hoare, "Quicksort," *The computer journal*, vol. 5, no. 1, pp. 10–16, 1962.

[4] S. G. Akl, *Parallel sorting algorithms.*

0.4emAcademic press, 2014, vol. 12.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres,

V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine *et al.*, "Open mpi: Goals, concept, and design of a next generation mpi implementation," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface: 11th European PVM/MPI Users' Group Meeting Budapest, Hungary, September 19-22, 2004. Proceedings 11.*

0.4emSpringer, 2004, pp. 97–104.

[6] D. Buttlar, J. Farrell, and B. Nichols, *Pthreads programming: A POSIX standard for better multiprocessing.*

" O'Reilly Media, Inc.", 1996.

[7] 4minus

Wikipedia contributors, "C Data Types," Wikipedia, The Free Encyclopedia, 2023, Accessed on June 2, 2023. [Online]. Available: https://en.wikipedia.org/wiki/C_data_types

[8] 4minus

——, "Fisher-Yates shuffle," Wikipedia, The Free Encyclopedia, 2023,

Accessed on June 2, 2023. [Online]. Available:
https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

[9] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer*, vol. 41, no. 7, pp. 33–38, 2008.