Project Report - Parallel Mergesort

Zoran Pecic and Anemone Kampkötter

9th January 2018

1 Introduction

Sorting is a fundamental problem in the field of Computer Science. It is a necessary and useful paradigm in most kinds of software and is important in many contexts. It must sometimes be done over huge data sets. Sorted data can be manipulated and handled easier in order to proceed any other algorithm on big data. Performance is very important, especially when we work with big data issues or real-time applications. Some sorting algorithms are good parallel algorithms, among these the algorithm Merge Sort can be found. Hence, in this report, we first want to describe the Mergesort algorithm in general and then represent our approach of parallelizing it in order to achieve a better performance for the sorting routine.

In the following we will first present the sequential version of Mergesort, explain its behaviour and its time complexity, followed by some ideas how to optimize the sequential algorithm. Then we will go over to our optimization, which consists of a parallelized version of Mergesort. In our attempt we used the OpenMP framework in order to parallelize the algorithm written in C++. We will give a description and an explanation of how we implemented it, how it can be executed and which performance we achieved. Our results and other solutions are discussed finally.

2 Mergesort

Merge sort is an algorithm based on the Divide-and-Conquer-principle. It is a fast and reliable sorting algorithm [Rad11]. As it works with the Divide and Conquer paradigm, we divide our main problem into sub-problems which are solved recursively. After that we use the all the resulting solutions for solving the main problem. Divide and Conquer is a good method to parallelize a problem as we can solve the sub-problems using multiple processors simultaneously.

2.1 Sequential Mergesort Algorithm

We divide a given array of length n into two sub-arrays of equal sizes and apply merge sort recursively on each of the sub-arrays. This is done until the last level is reached where every sub-array consists of just one element. Then we merge the so given sub-arrays with each other until we get one sorted array of length n [Rad11].

2.2 Pseudocode for the Sequential Mergesort

```
MergeSort(array, start, end)
        if (end - start) == 1
                return
        middle = (start+end)/2
        MergeSort(array, start, middle)
        MergeSort(array, middle, end)
        Merge(array, start, middle, end)

Merge( array, start, middle, end)
        tempArray[end-start]
        tempIndex = 0
    left = start
        right = middle
        while left < middle && rigth <
            end  do
                if array[left] < array[
                    right]
                    tempArray[ tempIndex
                        ++ ] = array[left
                    ++]
            else
                    tempArray[
                        tempIndex++]
                        = array[right
                        ++]

    while left < middle do
            tempArray[tempIndex++] =
                array[left++]

    while right < end do
            tempArray[tempIndex++] =
                array[right++]

    //copying
    for(i = 0; i < (e-s); i++)
            array[s+i] = tempArray[i]
```

2.2.1 Description of the Pseudocode

In the beginning we call the MergeSort function which divides the input array into two parts, the left subarray starts with the index "start" and ends with the calculated index "middle" and the right one starts with index "middle" and ends with the index "end". For both parts we call MergeSort recursively. We proceed the division process until we reach (end - start == 1). When we have returned the left and the right part we call the function Merge. This Merge function first creates an array tempArray of size (end-start) in which we will put our result. We compare the values of all elements contained in the left and the right subarrays. We put the smaller values in the tempArray and increment the index of the subarray with the smaller element by one in order to go through all elements. This is done until one of the indices of the subarrays reaches its end. The next two while loops are just for copying the remaining elements which are bigger than the already sorted elements into the tempArray. In the end we have one for loop for copying the results from tempArray into our original array. In the following image [1.1] you can see a concrete example of how an array is divided and then merged into a sorted array in an ascending order.

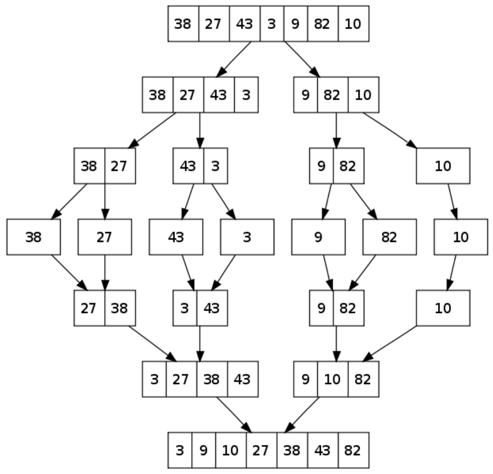


Figure 1: Example of Mergesort

2.3 Sequential Time Complexity

The sequential version of Merge Sort has the same time complexity for the best, the average and the worst case scenario which is O(n log n) [Rad11]. The complexity of log n stands for the dividing part in Mergesort as we divide the input array like a binary tree and the complexity of n results from the merging part where the left and the right part must be merged in order to be sorted.

2.4 Optimizations of the Sequential Algorithm

The sequential version of Mergesort can be optimized by removing the recursion and concentrate on a specific utilization of the cache memory. This is succeeded by changing the architecture of the CPU chip and reducing the number of cache misses. This is a low level kind of optimization [LL99].
Another approach of optimizing Merge Sort or any other sorting algorithm in general uses genetic algorithms and a classifier system in order to create "hierarchically-organized" hybrid sorting algorithms, which are implemented by library generators. It was shown that the performance of this approach is better than usual sorting algorithms. The focus lays in applying a more effective empirical search to the

generation of sorting routines, which depend much on the input sizes and architectural standards of the computing device. This search proceeds by generating different kinds of sorting trying to find the best parameter values of a sorting algorithm [XP05]. These are optimizations based on the sequential version of Mergesort. In the next parts of this report we present our version of optimization which deals with a parallelization of the sequential sorting problem.

3 Parallel Mergesort

It was proven theoretically that a parallel Mergesort algorithm is possible to be executed in O(log n) time [Col88]. We want to try to prove this by investigating Mergesort running on systems with OpenMP.

3.1 Technological Details

For our parallelization we used the OpenMP framework which allows us to develop multi-threaded and shared memory parallel applications as being a higher level programming model. This framework is a standard for C/C++ and Fortran compilers. The library provides us with basic routines for parallel programming. Every program starts like a single thread application, but in each parallel region which the programmer created with compiler directives the instruction flow and data stream are shared among multiple threads.

OpenMP provides us with directives for specifying parallel regions, work sharing, synchronization and data environment. For every defined parallel region, a master thread creates a team of threads exclusively. They are dynamically assigned to the parallel regions, work simultaneously in the parallel region and are synchronized altogether in the end of the region [Bla17].

Some advantages of OpenMP are that the same code can easily be modified in order to run sequentially, when the preprocessor OpenMP directives are just commented, and that OpenMP maps naturally on a multicore architecture. Besides, OpenMP can add parallelism incrementally and the compiler can optimize the code [and17].

3.2 Our Implementation

The most time consuming part in the sequential version of Mergesort is the merge function which gives us a time complexity of O(n). This time occurs as in every level of the resulting tree, which we achieve through dividing the array recursively, we need to compare every element which means that we must go through the whole array. Our solutions divide the array into P parts, where P is the number of physical cores available on the machine which is any number that is a power of 2. Then every part is assigned to one processor in order to compute the sequential Mergesort. This means that here we will theoretically have a time complexity of O(n/P log(n/P)), if n is the length of the array.

In the following we represent the pseudocode for the parallel version of MergeSort:

```
ParallelDivide( array , number_of_cores ,
   length)
       part = length / number_of_cores

       for i = 0 until i <
           number_of_cores do in
           parallel
               MergeSort(array , i*part ,
                   (i+1)*part)
       end of for
       number_of_cores /= 2



       while number_of_cores > 0 do
               part = length/
                   number_of_cores
               for i = 0 until i <
                   number of cores do in
                    parallel
                    start = i*part;
                    end = (i+1)*part
                    middle (s+e)/2
                        Merge(array , star
                                          , middle , end
                                          )
                            end of for
                            number_of_cores/=2
                  end of while
```

After the first for-loop we have P parts of the array and every part is sorted. Now we need to merge this parts which can be done by using half of the physical cores and calculating again number of elements of the array which will be assigned to every processor for merging. We merge like that until number_of_cores reaches 0. The result is one correctly sorted array.

We implemented this parallel solution in C++. For the first and the second for-loop we use the OpenMP clause "#pragma omp parallel for num_threads (number_of_threads)". The directive "#pragma omp parallel" starts the parallel region. The "for" - clause helps us to execute the for-loop with multiple threads by assigning different portions of the loop to each thread. With the appending "num_threads"-clause we define the number of threads to use in the parallel part.

3.3 Parallel Time Complexity

In our implementation we first the divide array into subarrays whose length is n/number_of_cores. Then we create threads in the for loop, so that every thread executes the MergeSort sequential algorithm whose complexity is O(n log(n)), but in our case it should be O(n/number_of_cores log(n/number_of_cores)). We run this on a machine with 4 physical cores with 2 and 4 threads which means for this part in our case the time complexity is O(n/2 log(n/2)) and O(n/4 log(n/4)) respectively.
In the next part of the algorithm, in the while loop, we create twice bigger subarrays than in the previous iteration and we create a twice smaller number of threads until we reach that the number of threads is equal to 0. Then every time when we execute our algorithm in the last iteration of the while loop we will have one thread which will merge the two subarrays which are two halves of original array. This means that this merging will be done sequentially with a linear complexity O(n).

4 Guide how to Execute our Solution

4.1 Requirements

The OpenMP API is a compiler extension for the programming languages C and C++ to enable parallel programming. In order to execute our code you will need these requirements:

1. Some C++ compiler
2. Included libraries:
   (a) OpenMP
   (b) Standard input/output
   (c) Math
   (d) Standard library C++

These libraries are already included as headers in the beginning of our code. As they are part of the C++ compiler, it is not necessary to install anything else apart from the compiler.

4.2 Compiling

In our source files we included a Makefile called "makefile" which contains the following lines:

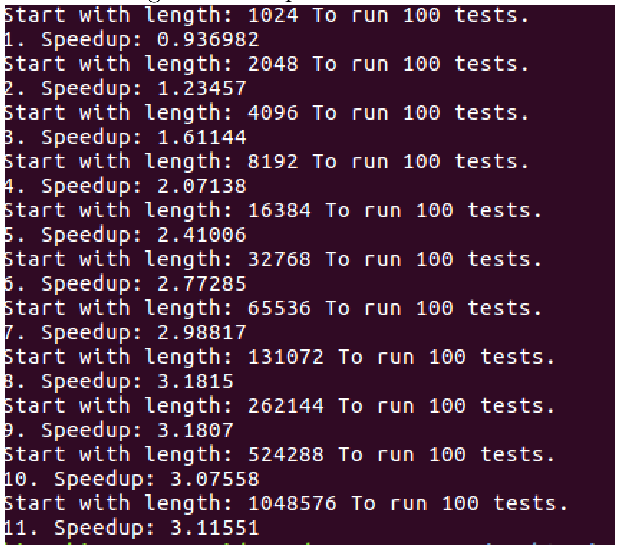
```
compile:
g++ -fopenmp -lm -O3 -o solution solution.cpp
```

We had to specify the command "fopenmp" in the makefile in order to link the OpenMP pragmas with the compiler. Now we can compile our solution by opening a terminal in the directory where the complete source file is stored and by typing the command "make compile". After that we can run it on the terminal with the command "./solution".

4.3 Exemplary Output

Here we show how an output after executing our code will look like:

Figure 2: Output with 100 tests



```
Start with length: 1024 To run 100 tests.
1. Speedup: 0.936982
Start with length: 2048 To run 100 tests.
2. Speedup: 1.23457
Start with length: 4096 To run 100 tests.
3. Speedup: 1.61144
Start with length: 8192 To run 100 tests.
4. Speedup: 2.07138
Start with length: 16384 To run 100 tests.
5. Speedup: 2.41006
Start with length: 32768 To run 100 tests.
6. Speedup: 2.77285
Start with length: 65536 To run 100 tests.
7. Speedup: 2.98817
Start with length: 131072 To run 100 tests.
8. Speedup: 3.1815
Start with length: 262144 To run 100 tests.
9. Speedup: 3.1807
Start with length: 524288 To run 100 tests.
10. Speedup: 3.07558
Start with length: 1048576 To run 100 tests.
11. Speedup: 3.11551
```

5 Performance Evaluation

5.1 Hardware Architecture

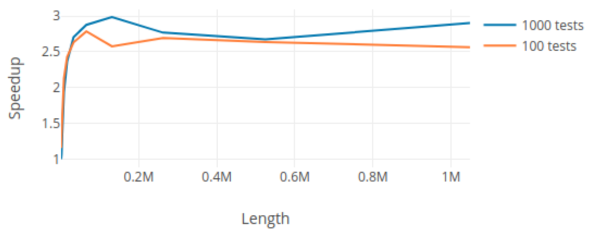
Our solution was tested on an Intel Core i5 CPU architecture of the 8th generation, specifically said on an i5-8250. This CPU has four physical cores, can run eight threads with hyper-threading and has the cache memory "smartCache" with 6 MB of memory space. Each core has a base frequency of 1.60 GHz and a maximum turbo frequency of 3.40 GHz. We used the g++ compiler version 5.4.0 which runs the OpenMP 4.0 version under GNU/ Linux.

5.2 Results

We achieved the following results by executing our code with a starting input size of 1024, which is doubled after n tests of sorting. After executing all tests and measuring their execution times, we compute the average execution time for the sequential and the parallel algorithms. In the end we compute the speedup by diving the average sequential execution time by the average parallel execution time.
In the following table we present some speedup results with four physical cores.

Speedup	Tests	
Length	100	1000
1024	1.15346	0.998284
2048	1.2505	1.06393
4096	1.60995	1.3783
8192	2.13274	1.93649
16384	2.43048	2.36607
32768	2.63227	2.70586
65536	2.7857	2.87678
131072	2.57658	2.98492
262144	2.69247	2.7678
524288	2.63626	2.67351
1048576	2.56323	2.90169

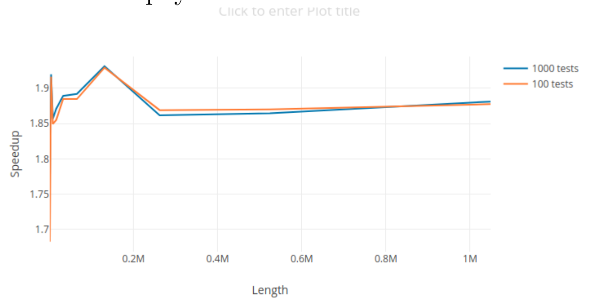
Figure 3: Diagram speedup results for with our algorithm on four physical cores



In the following table we present some speedup results with two physical cores.

Speedup	Tests	
Length	100	1000
1024	1.68289	1.77519
2048	1.81677	1.84135
4096	1.91619	1.91953
8192	1.84927	1.85682
16384	1.85508	1.87091
32768	1.8847	1.88917
65536	1.88459	1.89199
131072	1.92897	1.93105
262144	1.86896	1.86174
524288	1.87002	1.8646
1048576	1.87747	1.88111

Figure 4: Diagram speedup results for with our algorithm on two physical cores



5.3 Explanation of our Results

For the sequential and parallel algorithm we used the omp_get_wtime() function because it is the standard function in an OpenMP environment for measuring the wall clock time used for executing the program. We measure the average execution times for the sequential and the parallel algorithms by running a number of n tests. Then we divide both execution times with each other in order to get the speedup.
We get the speedup by dividing the array into blocks in the first for loop and in every iteration of the while loop when we merge the subarrays with more than one thread. The merge function itself is done sequentially but it can be done by two threads simultaneously, one working on the left side, the other one working on the right side.
In the last iteration of the while loop we now have just one remaining thread which means the merging is done by only one processing unit. Hence, the time complexity here is only O(n) as in the sequential version which is the bottleneck of our solution.
In the end we have some acceleration results which are not that bad. For two cores speedup is around 1.8 times and for the four cores speedup is around 2.7 times.

5.4 Other solutions

In our researches we found another interesting solution which is described in “Parallel merging sort with Double Merging” [Uya14]. This solution does the merging part in parallel using two threads. One thread compares the values of the array elements starting from the beginning of the left part and the beginning of the right part. After every comparison, it puts the smaller element into a temporary array and moves forward through the whole array. The second thread compares the values of the array elements starting from the last element of the left subarray and the last element of the right one. But this thread takes the bigger element of both compared values and puts it into the temporary array, starting in the end and going from the right to the left. In the end we will have a sorted array with two threads which gives us a time complexity of O(n/2), if n is the length of the input array. We tried to implement this solution, but we could not manage to get an appropriate execution time.
In another solution only the dividing part was parallelized. This solution uses parallel sections to assign the recursive dividing calls to the threads. We tried this solution but when running the code the solution did not give us a speedup. This was because the merging part was still sequential, hence the time complexity for the merging part is still O(n). For this reason we put our main focus on parallelizing the merge function as well. Most of other solutions we found refer to a MPI solution and concentrate on communication among processes.

6 Discussion

While implementing we had problems with measuring the execution time, we measured first the amount of CPU time which did not give us the result we wanted. Then we changed to measuring the wall clock time which is also affected from many external factors. In order to get a reliable value of the execution time we needed many measurements, hence, we run a number of n tests and calculated the average execution times before calculating the speedup. Still an open problem is that we were not able to parallelize the heavy-weight merge function. As this is the bottleneck in our implementation, we could not achieve a better performance than we presented in our results.

To check if our solution is correct we first checked the result of sorting on some smaller arrays and printed all elements of the array. Then we implemented the function checkArray() for checking the result of sorting which checks if every element of the array is smaller or equal to the next one. With this function it is easier to verify the correctness of our code's behaviour for larger array sizes.
With OpenMP we were not so flexible and did not have much control when we tried to fine-tune the code to achieve a higher speedup. We tried to implement "Parallel Merge Sort with Double Merging"[Uya14] but we did not get a speedup even if we allow nesting of sections and threads with "omp_set_nested(1)" it gave us even a worse speedup.
When we had the idea of parallelizing the merge function, we expected a speedup of more than 3 times. Our results show that we achieved a speedup of around 2, which is still satisfying as it can lead to a great performance improvement, especially when working with large data sets.

References

[Col88] Richard Cole. "Paralle Merge Sort". In: *SIAM J. Comput., 17(4), 770–785* (1988). DOI: `http://epubs.siam.org/doi/10.1137/0217049`.

[LL99] Richard E. Ladner and Anthony LaMarca. "The Influence of Caches on the Performance of Sorting". In: (1999). DOI: `https://cr.yp.to/bib/1999/lamarca-sorting.pdf`.

[XP05] Maria Jesus Garzaran Xiaoming Li and David Padua. "Optimize Sorting with Genetic Algorithms". In: (2005). DOI: `https://www.eecis.udel.edu/~xli/publications/genesort.pdf`.

[Rad11] Atanas Radenski. "Shared Memory, Message Passing, and Hybrid Merge Sorts for Standalone and Clustered SMPs". In: *CSREA Press (H. Arabnia, Ed.)* (2011). DOI: `http://www1.chapman.edu/~radenski/research/papers/mergesort-pdpta11.pdf`.

[Uya14] Ahmet Uyar. "Parallel Merge Sort with Double Merging". In: *IEEE Xplore* (2014). DOI: `http://ieeexplore.ieee.org.ep.bib.mdh.se/document/7036012/`.

[and17] Prof. Kunle Olukotun and. "OpenMP Lecture 14". In: *SIAM J. Comput., 17(4), 770–785* (2017). DOI: `https://web.stanford.edu/class/cs315b/lectures/lecture14.pdf`.

[Bla17] Lawrence Livermore National Laboratory Blaise Barney. "OpenMP". In: (2017). DOI: `https://computing.llnl.gov/tutorials/openMP/`.