PARALLEL SYSTEMS

ΠΑΝΕΠΙΣΤΗΜΙΟ ΔΥΤΙΚΗΣ ΑΤΤΙΚΗΣ
UNIVERSITY OF WEST ATTICA

DEPARTMENT OF COMPUTER ENGINEERING AND INFORMATION TECHNOLOGY

TASK 2A

Multisort

STUDENT / WORK DETAILS

NAME: ATHANASIOU VASILEIOS EVANGELOS REGISTRATION
NUMBER: 19390005
STUDENT SEMESTER: 11
STUDY PROGRAM: PADA

LABORATORY LEADER: IORDANAKIS MICHALIS
THEORY LEADER: MAMALIS VASILIOS

PARALLEL SYSTEMS

CONTENTS

1. Introduction 3

1.1 Purpose of exercise 3

1.2 Brief description of the problem being solved 3

2. Design 4

2.1 Description of the approach followed 4

2.2 Analysis of logic and methodologies 4

2.3 Description of data structures and algorithms 5

2.3.1 Data structures and variables 5

2.3.2 Generator program for producing 1D arrays 7

2.3.3 Multisort 7

2.3.4 Quicksort 7

2.3.5 Pivot Partition 7

2.3.6 Swap 8

2.3.7 Merge 8

3. Implementation 8

3.1 Reference to the basic functions of code 8

3.2 Explanation of parallel parts of code 9

3.3 Description of communication and synchronization between threads 10

4. Tests and Results 11

4.1 Reporting the execution conditions 11

4.2 Presentation of results in text format 12

4.2.1 Output_no_args.txt 12

4.2.2 Output.txt 12

4.2.3 Output_T1_N1000_L100.txt 12

4.2.4 Output_T1_N500000_L500.txt 13

4.2.5 Output_T1_N100000000_L1000.txt 13

4.2.6 Output_T2_N1000_L100.txt 14

4.2.7 Output_T3_N5000_L100.txt 14

4.2.8 Output_T4_N1000000_L500.txt 14

4.2.9 Output_T6_N500000_L500.txt 15

4.2.10 Output_T8_N100000000_L1000.txt 15

4.2.11 Output_T12_N50000000_L1000.txt 16

4.2.12 Output_T16_N100000000_L1000.txt 16

4.3 Efficiency analysis 16
4.3.1 Execution times of the parallel algorithm 16
4.3.2 Accelerations 21
4.3.3 Observations 26
5. Problems and Solutions 27
5.1 Reporting problems 27
5.2 Solutions tested and implemented 27
6. Conclusions 28
6.1 Recap 28

1. Introduction

1.1 Purpose of the exercise

The purpose of the exercise is to implement and evaluate the parallel 1D array sorting algorithm **multisort** , where it is an alternative to the **mergesort algorithm** . The parallel calculation is done using OpenMP which is at a low level the multithreaded parallel programming standard, with the logic of the fork - join parallel execution model .

Specifically, the programmer becomes familiar with the high level techniques provided by OpenMP , where it distinguishes the part of code that will be executed in parallel and understands the assignment of tasks to threads with the recursion property.

Finally, the exercise aims to highlight the execution times of the parallel algorithm, so that the speed - ups can be calculated and compared both in the case where the program is executed sequentially (with 1 thread) and in parallel (> 1 threads).

1.2 Brief description of the problem being solved

The algorithm solves the problem of sorting a 1D array of N integers A[1...N-1], using OpenMP for its parallel execution. This method is based on the **divide and conquer technique** , where the problem is decomposed into subproblems and the solutions of the subproblems are computed locally, so that they are finally collected for the best possible solution. Moreover, this is why it is an alternative to **mergesort** , where the array is decomposed into subarrays, sorted locally and finally merged into a single one which is also the final sorted one.

multisort algorithm initially divides the sequence to be sorted into four equal-sized parts, and continues by recursively applying the above separation procedure to each part. Upon completion of each recursive call, the four individual parts, which are returned to the main body sorted, are merged into a single sorted sequence in two steps (first they are merged – in parallel – two by two into two parts of twice the size, and then the two remaining parts are merged together).

Regarding performance, the following parameters are taken into account:

- Table size N
- Number of threads T
- Array limit LIMIT which also marks the end of the parallel calculation, where the well-known serial sorting algorithm **quicksort is executed**

2. Design

2.1 Description of the approach followed

The implementation is based on the distribution of calculations into tasks using OpenMP and recursion to achieve parallel processing, as 1 task is undertaken by only 1 thread . The approach is divided into distinct phases:

- **Assigning tasks to threads:** Using OpenMP and recursion, each task is taken over by an available active thread .
- **Exploiting the OpenMP feature :** In task allocation and synchronization with wait instructions .
- **Problem structure analysis:** Data is organized to facilitate parallel processing, minimizing communication between threads.

2.2 Analysis of logic and methodologies

The logic is based on the following:

1. **Partitioning the array A into 4 equal-sized parts:** The 1st phase of the algorithm is to partition the array into 4 equal-sized parts. This is achieved by calculating the appropriate indices, so that each part has approximately the same size N/4
2. **Recursive call of multisort for the 4 parts using OpenMP tasks and parallel execution by 4 available threads:** The 2nd phase of the algorithm is the recursive call for each of the 4 parts of the array using OpenMP tasks that assign each recursive call to a separate thread. Thus, the sorting of the 4 parts is done simultaneously
3. **Parallel merge by two into two parts of twice the size:** The 3rd phase is the parallel merge by two into two parts using the merge method .
4. **Merging the two remaining parts together:** The 4th phase is the merging of the two remaining parts into a single sorted array using the merge method again .
5. **Termination criterion check:** If, from the recursive call, the size of the array does not exceed the limit we set in the LIMIT parameter , then the recursion terminates and the serial sorting algorithm with the quicksort method is executed .

2.3 Description of data structures and algorithms

2.3.1 Data structures and variables

The following data structures and variables were used to solve the problem:

Variable Name	Variable Description
`int main ( int argc , character * argv [] )`	
int	
* A	Dynamic 1D array to be sorted
* Space	Dynamic 1D to be used as a temporary storage array for sorting A
threads	The number of threads to be created
size	The size of the table

I	Repetition indicator
double	
start_time	Start of measurement time of parallel processing of the algorithm
end_time	Timeout of the parallel processing of the algorithm
FILE	
* fpA_unsort	Output file for storing table A before it is sorted
* fpA_sort	Output file for storing table A after it has been sorted
`void multisort ( int * start , int * space , int size )`	
int	
*start	Index at the beginning of the array to be sorted
*space	Pointer to an auxiliary array of the same size as the original, used to merge the parts
size	Size of the array to be sorted
quarter	The size of the section to be separated
* startA	Index at the beginning of the 1st subsection
* startB	Index at the beginning of the 2nd sub-section
* startC	Index at the beginning of the 3rd sub-section
* startD	Index at the beginning of the 4th sub-section
* spaceA	Index to the beginning of the auxiliary table of the 1st sub-section
* spaceB	Index to the beginning of the auxiliary table of the 2nd sub-section
* spaceC	Index to the beginning of the auxiliary table of the 3rd sub-section
* spaceD	Index to the beginning of the auxiliary table of the 4th sub-section
`void quicksort ( int * start , int * end )`	
int	
*start	Index to the beginning of the part of the array to be sorted
*end	Index to the end of the part of the array to be sorted
* private	The guide element that will be placed in the correct sorting position in the table
`int * pivotPartition ( int * start , int * end )`	
int	
*start	Pointer to the beginning of the part of the array to be split
*end	Pointer to the end of the array section to be split
* private	The guide element that will be placed in the correct sorting position in the table
* i	Pointer pointing to a position before the one pointed to by pointer j

*j	Pointer pointing to a position after the one pointed to by pointer i
`void swap ( int * a , int * b )`	
int	
*a	Pointer to the first element to be swapped
*b	Pointer to the second element to be exchanged
temperature	Temporary storage space
`void merge ( int * startA , int * endA , int * startB , int * endb , int * space )`	
int	
* startA	Index to the first element of the first subarray (A)
* endA	Pointer to the last element of the first subarray (A)
* startB	Pointer to the first element of the second subarray (B)
* endb	Pointer to the last element of the second subarray (B)
*space	Index to temporary storage table
* i	Initialization index for sub-array A
*j	Initialization index for sub-table B
*k	Initialization index for the temporary storage table

2.3.2 Generator program for producing 1D arrays

This algorithm creates an N- dimensional array A of pseudorandom integers based on a range of values.

- **Code:**

```
for ( i = 0 ; I < size ? i ++)
{
        A [ i ] = rand ( ) % 199 - 99 ;
        A [ i ] = A [ i ] >= 0 ? A [ i ] + 10 : A [ i ] - 10 ;
}
```

- **Operation:**
    - We access the 1D array with a loop
    - We choose values from -99 to 99
    - We change the value by 10 depending on the sign

2.3.3 Multisort

The **multisort function** implements an alternative parallel sorting algorithm to **mergesort** . The algorithm divides the array into four equal-sized quarters , sorts each quarter recursively, and then merges the sorted quarters. When the quarter size is less than the LIMIT limit, the sequential **quicksort** algorithm is used for local sorting.

2.3.4 Quicksort

The **quicksort function** implements the recursive quicksort algorithm.

1. Selects a pivot element from the part of the table being sorted, that is, an element that is in the correct sort position.
2. It places all elements smaller than or equal to the guide to its left and all elements larger than it to its right, dividing the array into two subarrays.
3. It recursively repeats the process for the left and right subarrays until the complete sorting is achieved.

2.3.5 Pivot Partition

The **pivotPartition function** with the two-index technique performs the partitioning of the array around a pivot element .

1. The pivot is initially defined as the last element of the array section.
2. Moves all elements smaller than or equal to the pivot to the left side of the array, while larger elements remain on the right side.
3. Finally, the pivot is placed in its correct position, that is, between the smallest and largest elements.

2.3.6 Swaps

The **swap function** swaps the values of two variables .
indicated by the indices a and b .

2.3.7 Merge

merge function merges two sorted subsets (A and B) into a single sorted array. The merged array is first stored in the temporary array space and then returned to the original array.

3. Implementation

3.1 Reference to the basic functions of the code

The main functions of the code include:

- **Initialize array A with random values:**
  With a generator program referred to in chapter 2.3.2 Generator program for producing 1D arrays, the 1D array to be classified is created.
- **Multisort algorithm :**
  The algorithm includes:
  - Dividing the table into sub-tables: The original table is divided into 4 equal-sized parts
  - Parallel sorting of sub-arrays: Each sub-array is sorted simultaneously by a different thread. If the size of the sub-array is smaller than a user-defined threshold, then the sequential quicksort sorting algorithm is executed .
  - Parallel merging of sorted sub-arrays: After local sorting, the sub-arrays are merged to form the final sorted array.
- **Discrete task management:**
  To synchronize the discrete tasks that undertake recursive calls of the algorithm, synchronization mechanisms are also implemented for the correct parallel computation performed by the threads.
- **Time measurements:**
  For the execution of the algorithm, the start time of the parallel calculation and the end time are taken to calculate the speedups and performance of the algorithm.
- **Printout of results:**
  The final results are saved in a file.

3.2 Explanation of parallel parts of the code

The parallel sections are explained below:

- **multisort algorithm :**
  - Code:

```
#pragma omp parallel
{
   #pragma omp single
   multisort ( A , Space , size );
}
```

  - Operation:
    - The **#pragma omp parallel** activates parallel threads
    - The **#pragma omp single** ensures that only one thread belonging to the parallel region will execute the multisort algorithm .

- **Recursive call of multisort for the four parts using tasks for execution by different threads:**

- Code:

```
#pragma omp task firstprivate ( start , space , size )
multisort ( startA , spaceA , quarter );

#pragma omp task firstprivate ( start , space , size )
multisort ( startB , spaceB , quarter );

#pragma omp task firstprivate ( start , space , size )
multisort ( startC , spaceC , quarter );

#pragma omp task firstprivate ( start , space , size )
multisort ( startD , spaceD , size - 3 * quarter );

#pragma omp taskwait
```

- Operation:
  - The **#pragma omp task** defines a discrete task executed by a thread.
  - 4 tasks are created to sort each part of the array, with the last one having a larger size because the array may not be divided exactly into 4 equal-sized parts.
  - Each task will be executed independently, utilizing the available threads.
  - This mechanism offers more efficient execution in recursive calls of parallel algorithms.
  - The **firstprivate directive** specifies that each thread that executes the discrete task will have private variables (private) where they will also be initialized (firstprivate).
  - **The #pragma** directive **omp taskwait** synchronizes the threads so that they wait for all discrete tasks to complete before the parallel computation can proceed.
  - Without synchronization, we would have incorrect results.

- **Parallel merge by two into two parts of twice the size:**
  - Code:

```
#pragma omp task firstprivate ( start , space , size )
merge ( startA , startA + quarter - 1 , startB , startB + quarter - 1 ,
spaceA );

#pragma omp task firstprivate ( start , space , size )
merge ( startC , startC + quarter - 1 , startD , start + size - 1 , spaceC
);

#pragma omp taskwait
```

  - Operation:
    - OpenMP directives apply as mentioned above.

- 2 tasks are created to merge the sub-parts into two parts of twice the size
- We need task synchronization again with the **# pragma directive omp taskwait** so that we don't get incorrect results.

3.3 Description of communication and synchronization between threads

Communication and synchronization between threads is ensured by various mechanisms provided by OpenMP . The methods that are described below were used in the program:

- **Private Memory:**
  - Private initialized variables (firstprivate):
    For each task, the thread that undertakes it has its own copies of variables which are also initialized, e.g. *start , space , size* , etc.

  - Use in the program:

```
#pragma omp task firstprivate ( start , space , size )
```

    - Private variables ensure that each thread performs independent computational operations

- **Synchronization Mechanisms**
  The correct operation of the program is based on synchronization of the discrete tasks, so that the threads wait for all tasks to complete before the execution of the algorithm can proceed.

  - Use in the program:

```
#pragma omp taskwait
```

4. Tests and Results

4.1 Reporting of execution conditions

The execution is done for different array sizes N, thread numbers T and a limit LIMIT which is the termination criterion of the recursive call of the multisort algorithm .

The program is compiled via command line in a Linux environment , with the compiler GNU **gcc and** the **–fopenmp switch** for linking it with the **omp library . h .**

```
gcc –o omp_msort omp_msort.c - fopenmp
```

The program is executed via command line in a Linux environment and the user must pass 2 txt files as parameters , so that the table A to be sorted and the sorted one are saved respectively. Indicative execution command:

```
./omp_msort A _ unsort . txt A _ sort . txt
```

4.2 Presentation of results in text format

Output folder HYPERLINK "Output" *and the tables A _ unsort and A _ sort in the respective folders. To save space, not all results are presented in text format in this documentation. To be redirected to the output files, click on the link in the sub- headings below and respectively for the tables whose name is found both in the corresponding folders and in the text results.*

The program requires the user to pass 2 . txt output files with a name of his choice, in which the unsorted table A and the sorted one will be stored. In case the user does not enter the required number of parameters, the program terminates and a characteristic message is displayed

4.2.1 Output_no_args.txt

```
Usage : . / omp_msort A_unsort.txt A_sort.txt
```

To check the correctness of the algorithm, we first tested a small-sized table to confirm that classification is achieved.

4.2.2 Output.txt

```
Threads : 6
Matrix size: 36
Limit for quicksort : 4
-----------------------------------
Before sorting
-----------------------------------
The A has been stored in A_unsort / A_unsort.txt

-----------------------------------
After sorting
-----------------------------------
The A has been stored in A_sort / A_sort.txt

-----------------------------------
Multisort finished in 0.000301 sec .
```

```
-----------------------------------
```

In order to calculate the accelerations, we ran the algorithm sequentially.

4.2.3 Output_T1_N1000_L100.txt

```
Threads : 1
Matrix size: 1000
Limit for quicksort: 100
-----------------------------------
Before sorting
-----------------------------------
The A has been stored in A_unsort / A_unsort_T1_N1000_L100.txt

-----------------------------------
After sorting
-----------------------------------
The A has been stored in A_sort / A_sort_T1_N1000_L100.txt

-----------------------------------
Multisort finished in 0.000120 sec.
```

4.2.4 Output_T1_N500000_L500.txt

```
Threads : 1
Matrix size: 500000
Limit for quicksort: 500
-----------------------------------
Before sorting
-----------------------------------
The A has been stored in A_unsort /A_unsort_T1_N500000_L500.txt

-----------------------------------
After sorting
-----------------------------------
The A has been stored in A_sort /A_sort_T1_N500000_L500.txt

-----------------------------------
Multisort finished in 0.079525 sec.
-----------------------------------
```

4.2.5 Output_T1_N100000000_L1000.txt

```
Threads : 1
Matrix size: 100000000
Limit for quicksort: 1000
-----------------------------------
Before sorting
-----------------------------------
```

```
The A has been stored in A_unsort /A_unsort_T1_N100000000_L1000.txt

------------------------------------
After sorting
------------------------------------
The A has been stored in A_sort /A_sort_T1_N100000000_L1000.txt

------------------------------------
Multisort finished in 19.709039 sec.
------------------------------------
```

4.2.6 Output_T2_N1000_L100.txt

```
Threads : 2
Matrix size: 1000
Limit for quicksort: 100
------------------------------------
Before sorting
------------------------------------
The A has been stored in A_unsort / A_unsort_T2_N1000_L100.txt

------------------------------------
After sorting
------------------------------------
The A has been stored in A_sort / A_sort_T2_N1000_L100.txt

------------------------------------
Multisort finished in 0.000197 sec.
------------------------------------
```

4.2.7 Output_T3_N5000_L100.txt

```
Threads : 3
Matrix size: 5000
Limit for quicksort: 100
------------------------------------
Before sorting
------------------------------------
The A has been stored in A_unsort / A_unsort_T3_N5000_L100.txt

------------------------------------
After sorting
------------------------------------
The A has been stored in A_sort / A_sort_T3_N5000_L100.txt

------------------------------------
Multisort finished in 0.000470 sec.
------------------------------------
```

4.2.8 Output_T4_N1000000_L500.txt

```
Threads : 4
Matrix size: 1000000
Limit for quicksort: 500
-----------------------------------
Before sorting
-----------------------------------
The A has been stored in A_unsort /A_unsort_T4_N1000000_L500.txt

-----------------------------------
After sorting
-----------------------------------
The A has been stored in A_sort /A_sort_T4_N1000000_L500.txt

-----------------------------------
Multisort finished in 0.057440 sec.
-----------------------------------
```

4.2.9 Output_T6_N500000_L500.txt

```
Threads : 6
Matrix size: 500000
Limit for quicksort: 500
-----------------------------------
Before sorting
-----------------------------------
The A has been stored in A_unsort /A_unsort_T6_N500000_L500.txt

-----------------------------------
After sorting
-----------------------------------
The A has been stored in A_sort /A_sort_T6_N500000_L500.txt

-----------------------------------
Multisort finished in 0.021820 sec.
-----------------------------------
```

4.2.10 Output_T8_N100000000_L1000.txt

```
Threads : 8
Matrix size: 100000000
Limit for quicksort: 1000
-----------------------------------
Before sorting
-----------------------------------
The A has been stored in A_unsort /A_unsort_T8_N100000000_L1000.txt
```

```
-----------------------------------
After sorting
-----------------------------------
The A has been stored in A_sort /A_sort_T8_N100000000_L1000.txt

-----------------------------------
Multisort finished in 6.076829 sec.
-----------------------------------
```

4.2.11 Output_T12_N50000000_L1000.txt

```
Threads : 12
Matrix size: 50000000
Limit for quicksort: 1000
-----------------------------------
Before sorting
-----------------------------------
The A has been stored in A_unsort /A_unsort_T12_N50000000_L1000.txt

-----------------------------------
After sorting
-----------------------------------
The A has been stored in A_sort /A_sort_T12_N50000000_L1000.txt

-----------------------------------
Multisort finished in 2.945010 sec.
-----------------------------------
```

4.2.12 Output_T16_N100000000_L1000.txt

```
Threads : 16
Matrix size: 100000000
Limit for quicksort: 1000
-----------------------------------
Before sorting
-----------------------------------
The A has been stored in A_unsort /A_unsort_T16_N100000000_L1000.txt

-----------------------------------
After sorting
-----------------------------------
The A has been stored in A_sort /A_sort_T16_N100000000_L1000.txt

-----------------------------------
Multisort finished in 6.174742 sec .
-----------------------------------
```

4.3 Efficiency analysis

4.3.1 Execution times of the parallel algorithm

The times recorded for different number of threads T and array size N are presented in the diagrams below. Specifically, the results data are:

- **Number of threads:** 1, 2, 3, 4, 6, 8, 12 and 16
- **Table Size:** 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000, 50000000, 100000000

Threads	Total Time
1	0.00012
2	0.000197
3	0.000504
4	0.001087

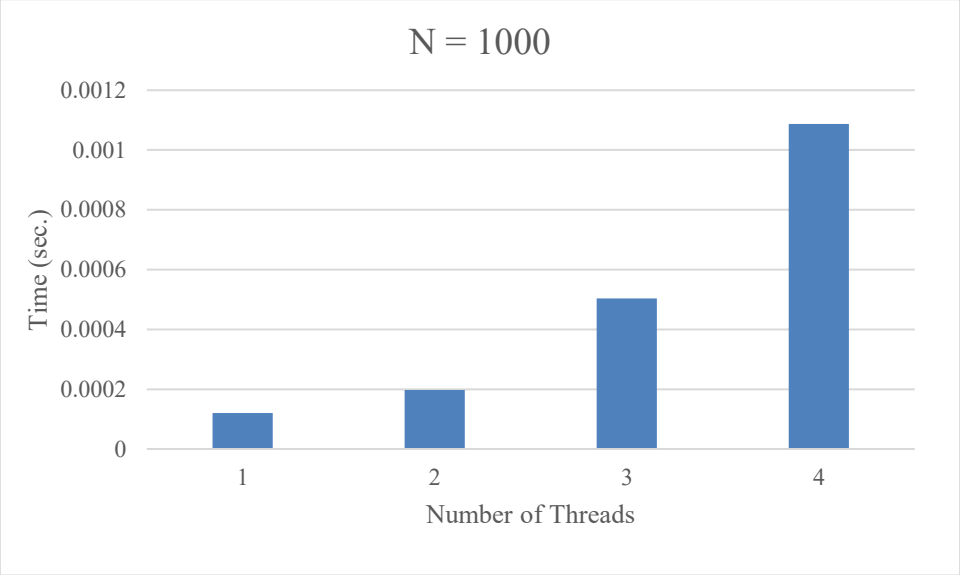

Figure 1. Execution times of the parallel algorithm for N = 1000

Threads	Total Time
1	0.000654
2	0.000459
3	0.00047
4	0.00194

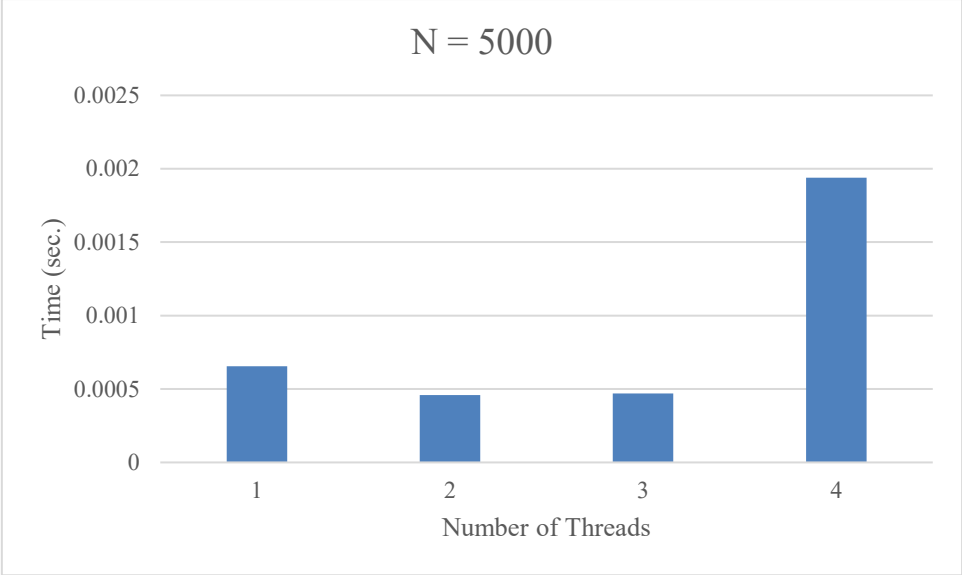

Figure 2. Execution times of the parallel algorithm for N = 5000

Threads	Total Time
1	0.001412
2	0.000955
3	0.00138
4	0.003595

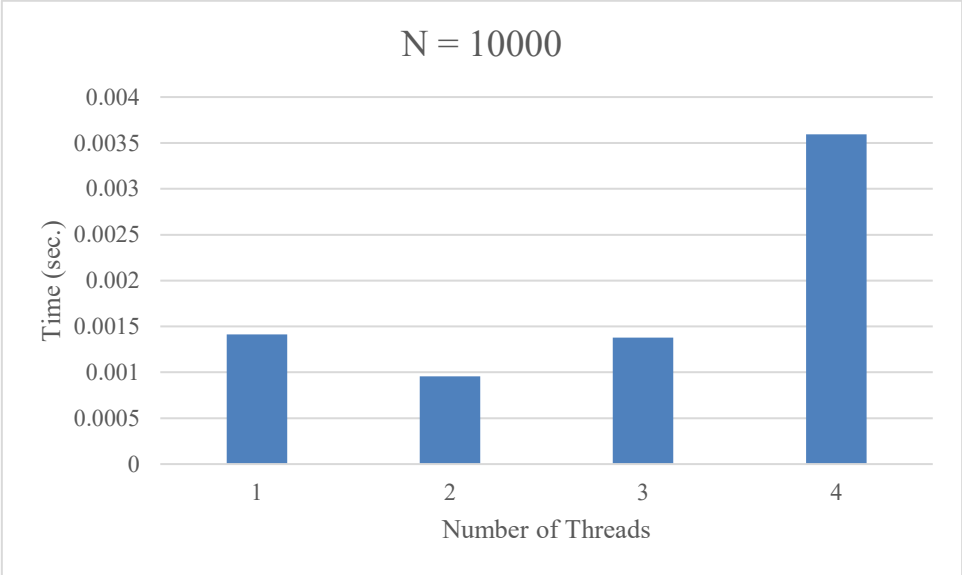

Figure 3. Execution times of the parallel algorithm for N = 10000

Threads	Total Time
1	0.015074
4	0.005436
6	0.005173
8	0.005402

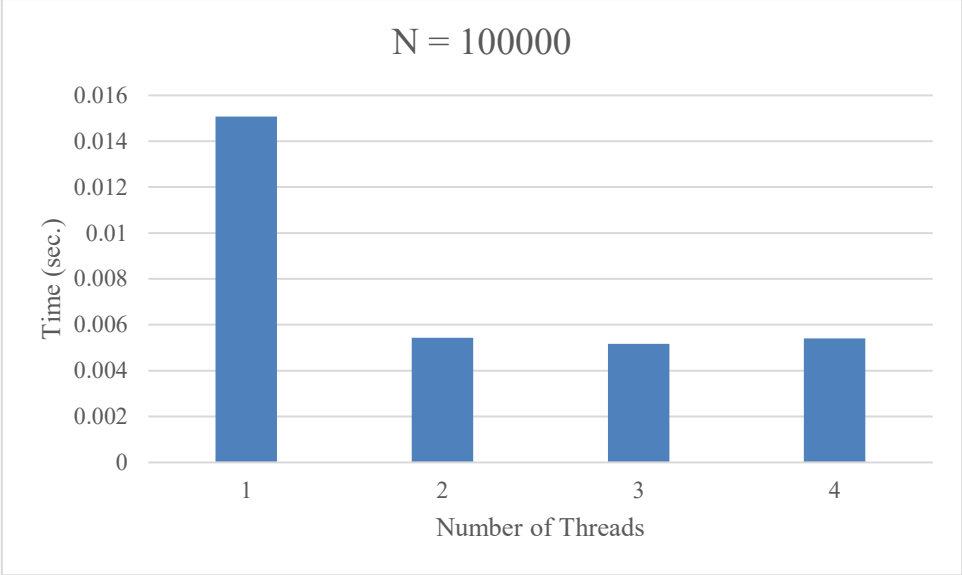

Figure 4. Execution times of the parallel algorithm for N = 100000

Threads	Total Time
1	0.079525
4	0.030407
6	0.02182
8	0.025397

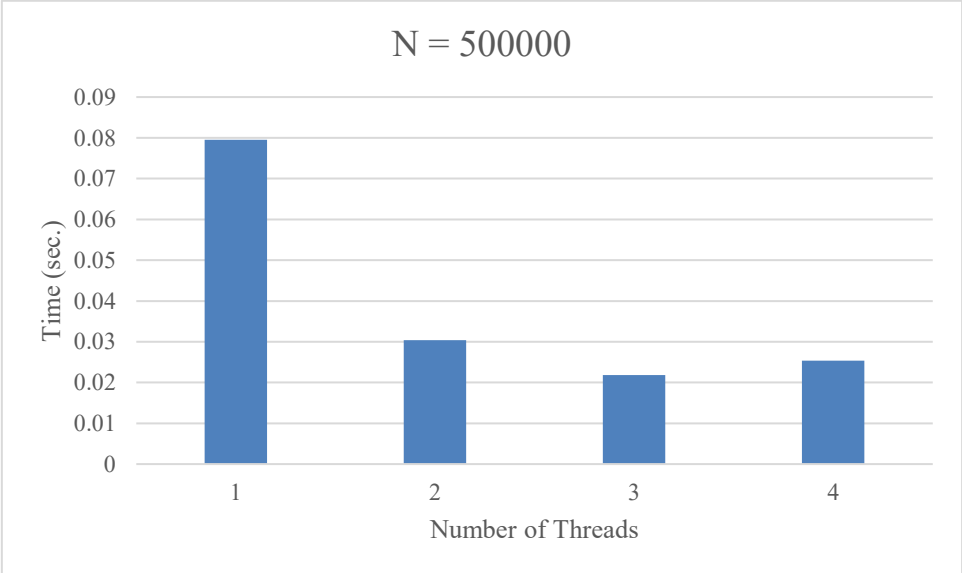

Figure 5. Execution times of the parallel algorithm for N = 500000

Threads	Total Time
1	0.151469
4	0.05744
6	0.057255
8	0.060294

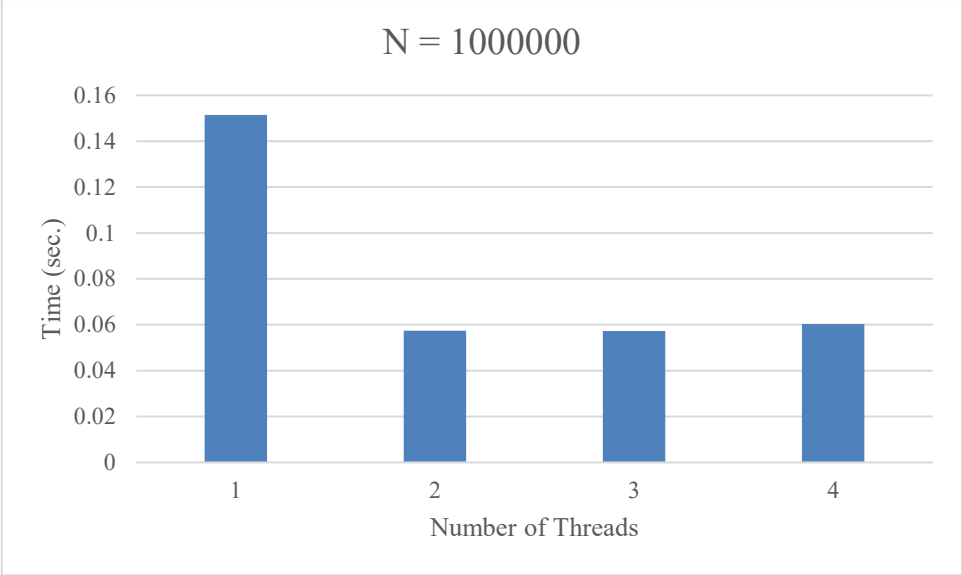

Figure 6. Execution times of the parallel algorithm for N = 1000000

Threads	Total Time
1	1.795719
8	0.515042
12	0.525958
16	0.607034

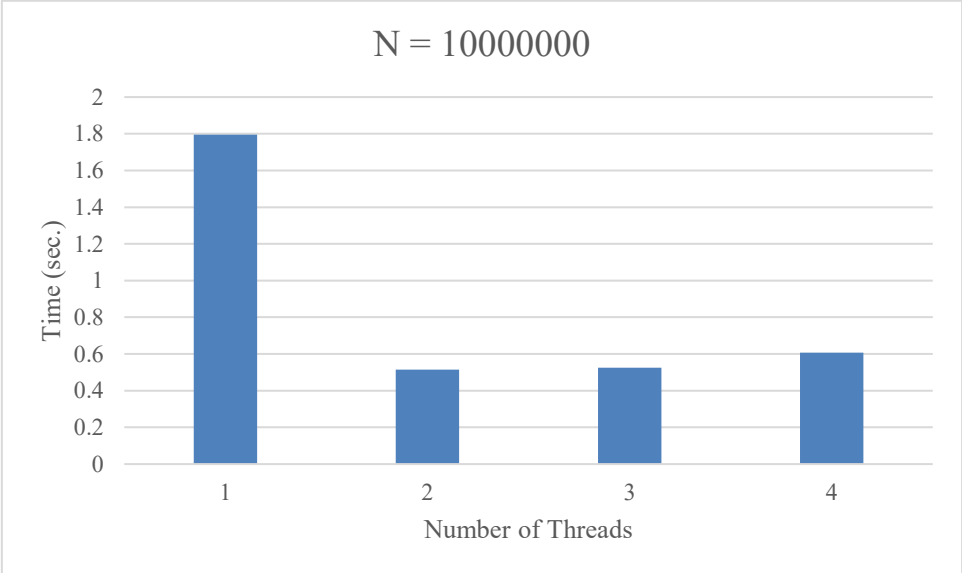

Figure 7. Execution times of the parallel algorithm for N = 10000000

Threads	Total Time
1	9.168969
8	3.122941
12	2.94501
16	3.050295

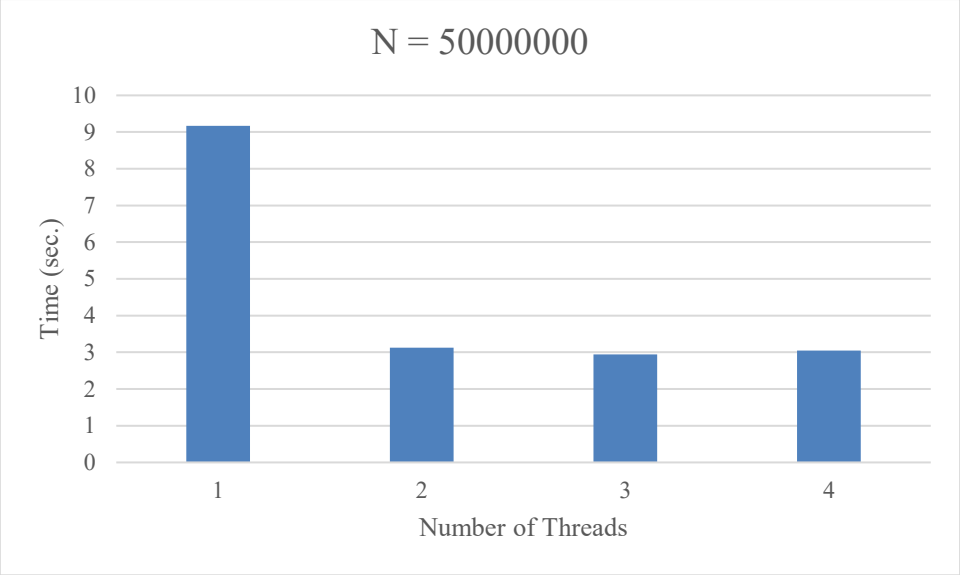

Figure 8. Execution times of the parallel algorithm for N = 50000000

Threads	Total Time
1	19.70904
8	6.076829
12	6.154808
16	6.174742

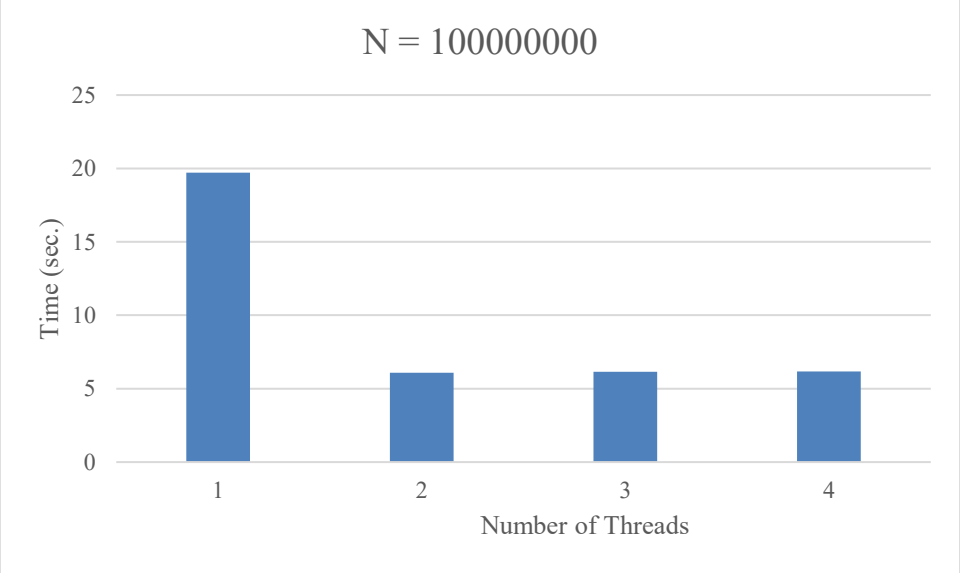

Figure 9. Execution times of the parallel algorithm for N = 100000000

4.3.2 Accelerations

Accelerations are calculated by the following mathematical formula:

$$\text{Speedup} = {}^{t_s}/_{t_n}$$

t_s: The execution time of the sequential program, i.e., execution with a 1 thread
t_n: The execution time of the parallel program, i.e., execution with n threads

Source: Lesson 2 – Parallel Computing – Introduction to Parallel Computing – p. 7

The accelerations are presented in the diagrams below. Specifically, the results data are:

- **Number of threads:** 1, 2, 3, 4, 6, 8, 12 and 16
- **Table Size:** 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000, 50000000, 100000000

Threads	Speed up
1	1
2	0.609
3	0.238
4	0.11

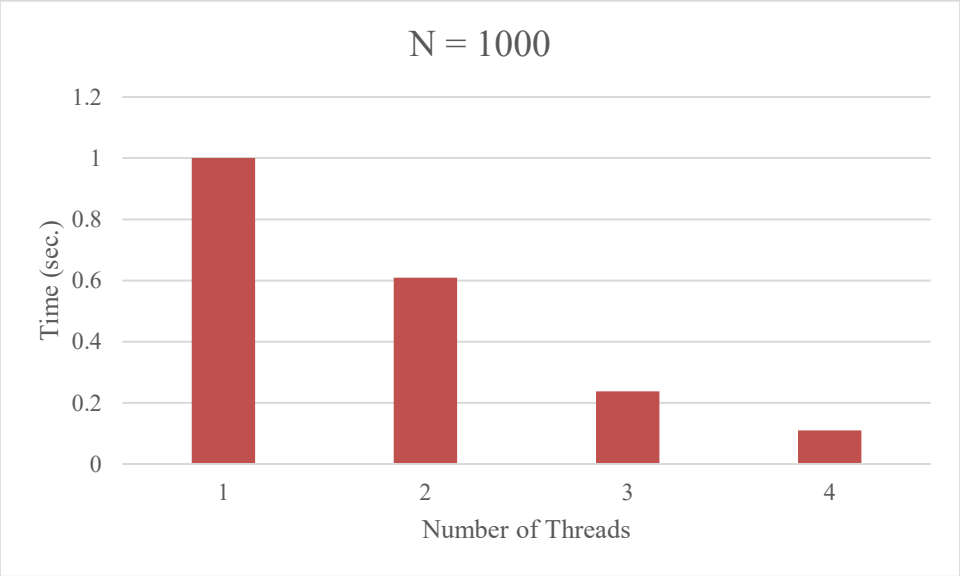

Figure 10. Speedups of the parallel algorithm for N = 1000

Threads	Speed up
1	1
2	1,426
3	1,392
4	0.338

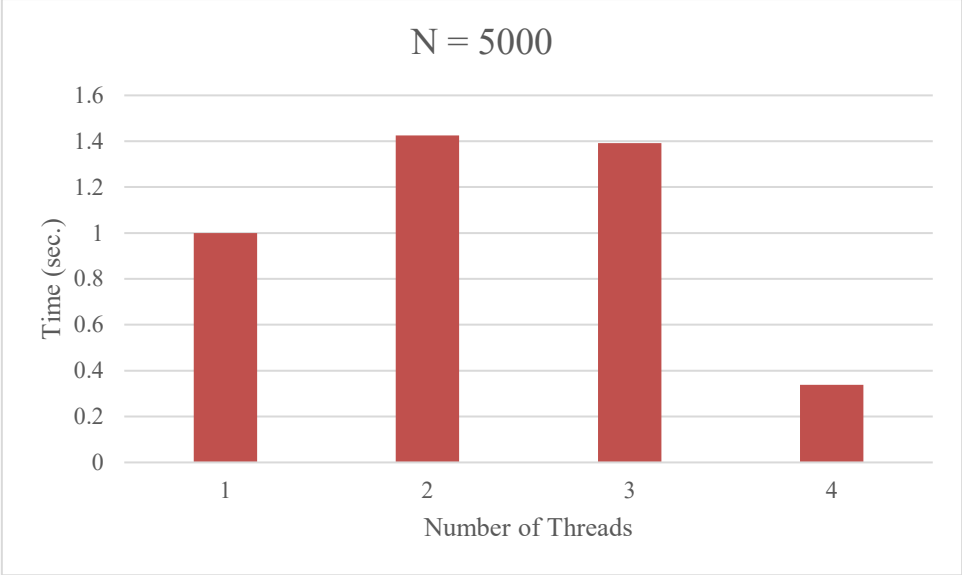

Figure 11. Speedups of the parallel algorithm for N = 5000

Threads	Speed up
1	1
2	1,479
3	1,023
4	0.393

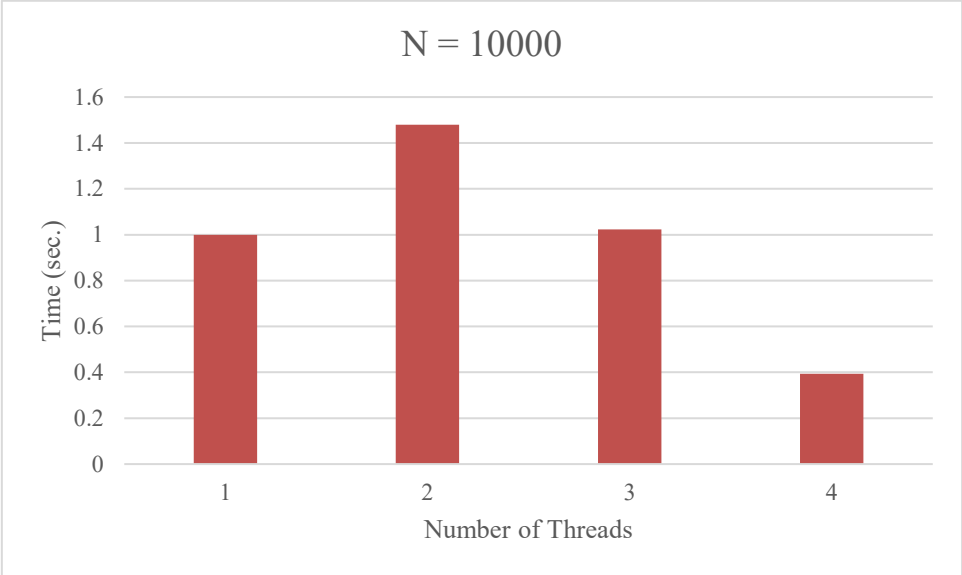

Figure 12. Speedups of the parallel algorithm for N = 10000

Threads	Speed up
1	1
4	2.77
6	2.91
8	2.79

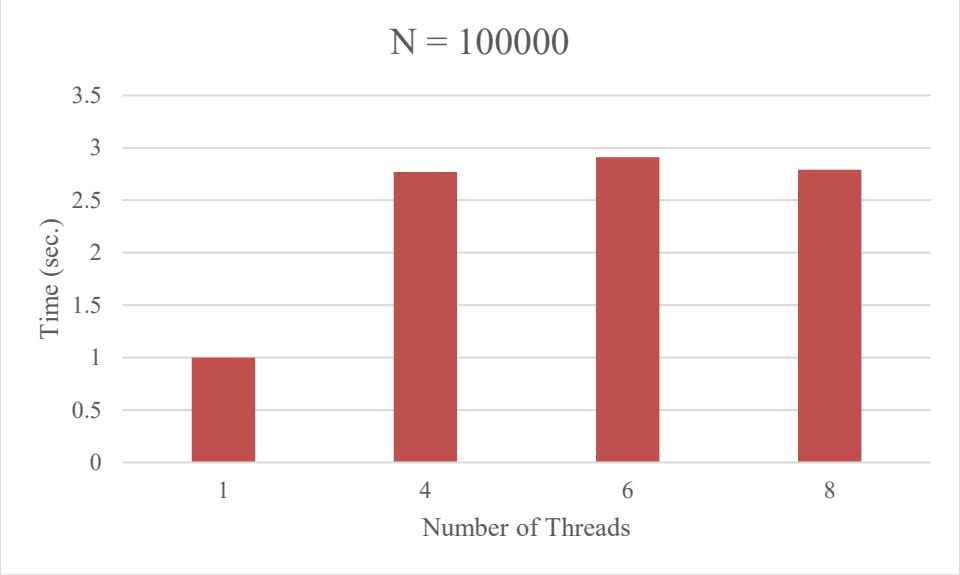

Figure 13. Speedups of the parallel algorithm for N = 100000

Threads	Speed - up
1	1
4	2.61
6	3.65
8	3.14

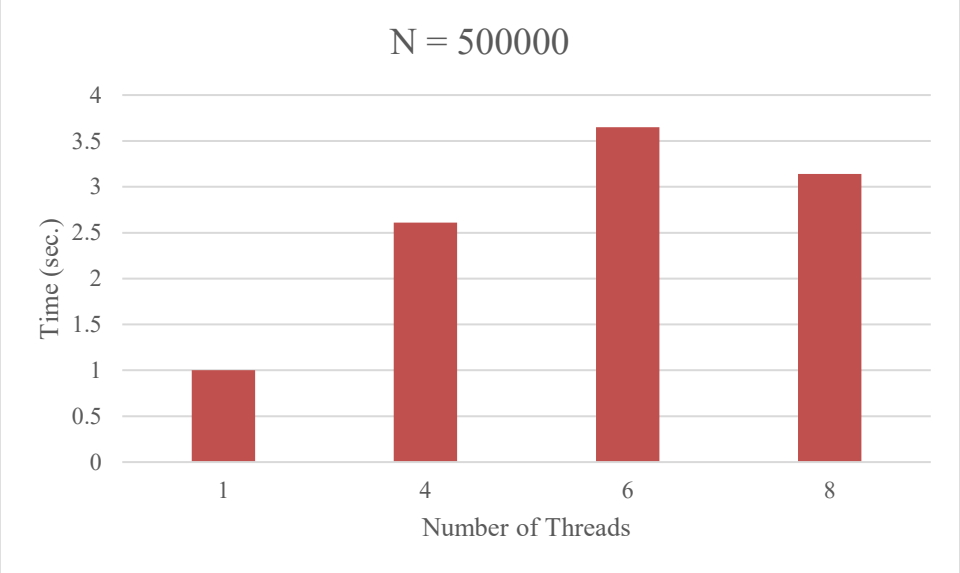

Figure 14. Speedups of the parallel algorithm for N = 500000

Threads	Speed up
1	1
4	2.63
6	2.64
8	2.51

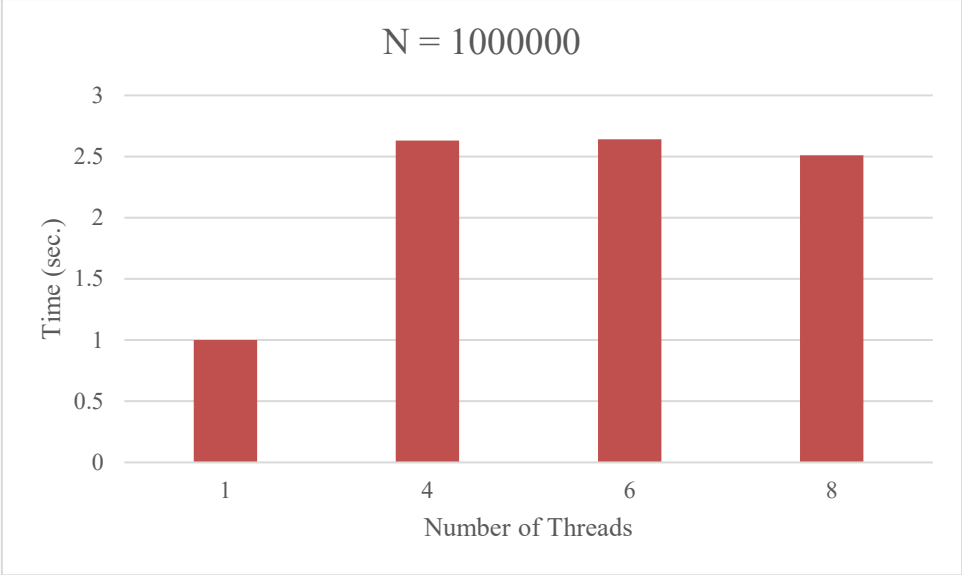

Figure 15. Speedups of the parallel algorithm for N = 1000000

Threads	Speed - up
1	1
8	3.48
12	3.41
16	2.96

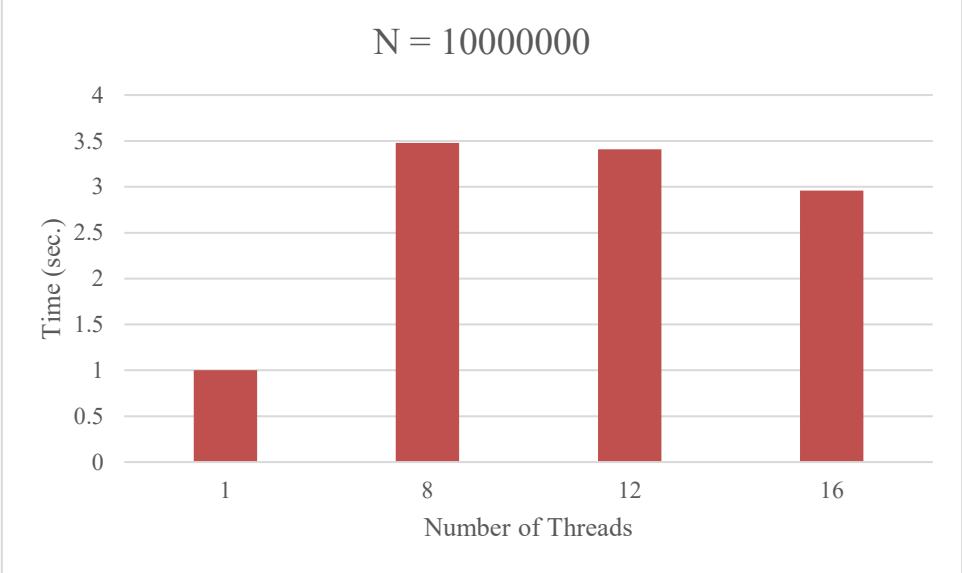

Figure 16. Speedups of the parallel algorithm for N = 10000000

Threads	Speed up
1	1
8	2.93
12	3.11
16	3.01

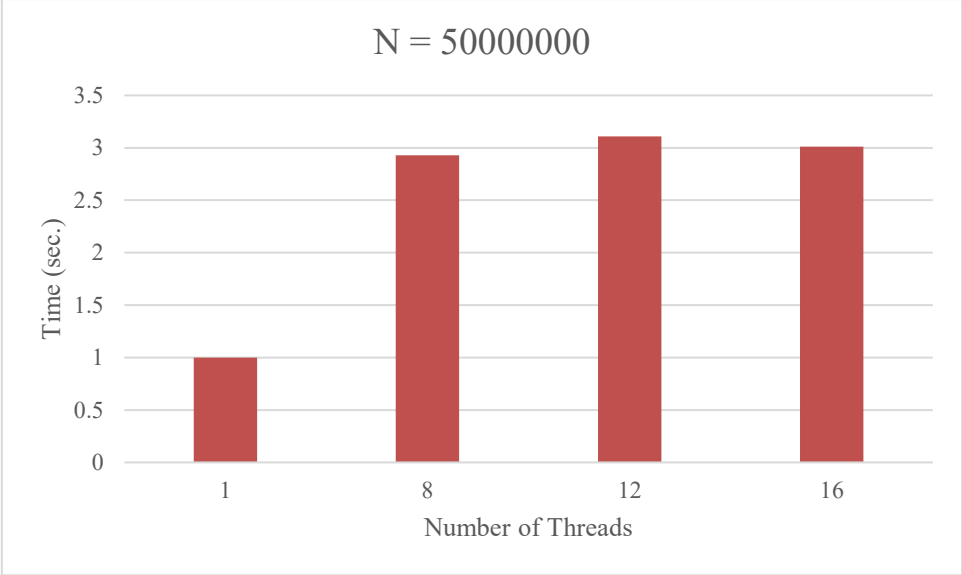

Figure 17. Speedups of the parallel algorithm for N = 50000000

Threads	Speed up
1	1
8	3.25
12	3.21
16	3.19

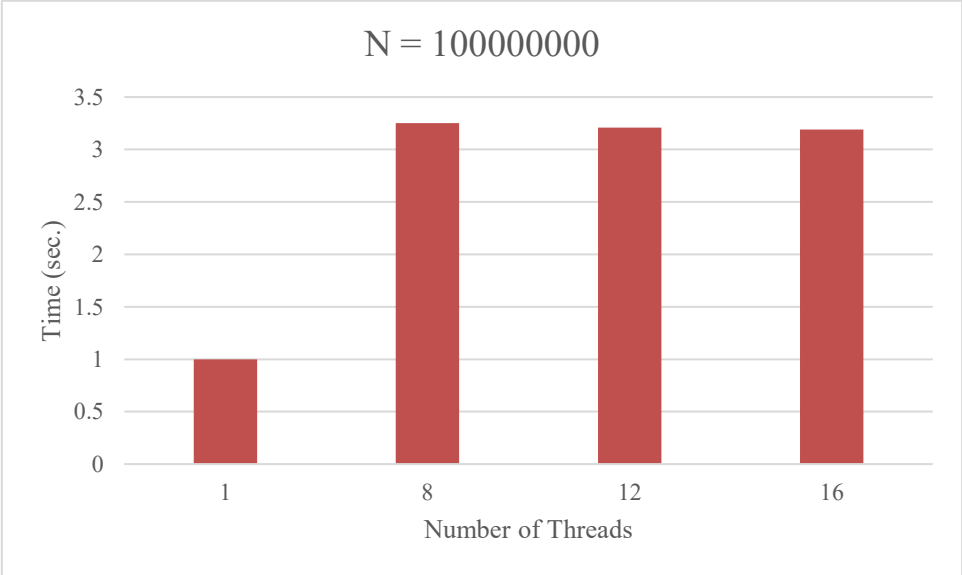

Figure 18. Speedups of the parallel algorithm for N = 100000000

4.3.3 Observations

We observe that for small N, parallelism does not yield better times than sequential. This is also due to the fact that creating and managing more threads can bring additional overhead rather than less. This time contradicts the benefits of parallel computing, as threads

compete for memory due to the small amount of data causing delays. There are also delays in synchronization, especially when we are dealing with discrete tasks undertaken by 1 thread and synchronization mechanisms with taskwait instructions .

For example, in the data we collected for **N = 1000** , we observe that execution times increase as we increase the number of active threads and therefore we do not achieve speedups > 1 sec , as the execution time of tasks with 1 thread is shorter than the corresponding one with 2, 3 or even 4 threads.

However, the benefits of parallel computing are noticeable for large N , as we observe in the data we collected for **N = 100000000** , as there we achieve better times with more threads and therefore better speedups. This is also due to the assignment of tasks to the available threads, where idle time due to the large volume of data is minimized.

Therefore, the benefits of parallel computing are more efficient for large volumes of data, as we do not have threads that remain inactive.

5. Problems and Solutions

5.1 Reporting problems

A problem was found in the management of indexes in the multisort and merge routines . Specifically, the role of the space index was not fully understood and how we would store the final sorted array. We also found a problem in the 4th ^{part} of the array, where if the array was not divided by exactly 4, then there were excess elements of the array that were not sorted.

also identified a problem with the execution times of parallel tasks when the working environment was WSL (Windows Subsystem for Linux), as it was taking longer to complete the parallel algorithm than expected.

5.2 Solutions tested and implemented

Finally, the space pointer pointed to a temporary storage array where we would use it to copy the sorted final array to the pointer pointing to the original one. As for the problem of sharing in the 4th ^{part} of the array, we changed the size to ensure that the 4th ^{part} would also have the excess elements in case the size is not divisible by 4:

```
multisort ( startD , spaceD , size - 3 * quarter);
```

For the times, we chose to change environment and go to a pure Linux distribution (Ubuntu) and the results were clearly better.

WSL

```
Threads : 6
Matrix size: 10000000
Limit for quicksort: 100
---------------------------------
```

```
Before sorting
-----------------------------------
The A has been stored in A_unsort.txt

-----------------------------------
After sorting
-----------------------------------
The A has been stored in A_sort.txt

-----------------------------------
Multisort finished in 0.670413 sec.
-----------------------------------
```

Linux

```
Threads : 6
Matrix size: 10000000
Limit for quicksort: 100
-----------------------------------
Before sorting
-----------------------------------
The A has been stored in A_unsort.txt

-----------------------------------
After sorting
-----------------------------------
The A has been stored in A_sort.txt

-----------------------------------
Multisort finished in 0.562977 sec.
-----------------------------------
```

After some research on the internet, we discovered the reason why parallel task execution times are longer in WSL :

OMP is incredibly slow in WSL2 due to filesystem boundary

https://github.com/JanDeDobbeleer/oh-my-posh/issues/1268

The performance issue with OpenMP in WSL 2 can often be attributed to the way WSL 2 handles the file system. WSL 2 uses a virtualized environment that interacts with the Windows file system , and this interaction can introduce significant latency when there are frequent I/O operations. This is especially noticeable with tools like OpenMP , where parallel threads may interact heavily with the file system for data storage or synchronization.

6. Conclusions

6.1 Recap

To recap, the parallel multisort algorithm with OpenMP tasks is an efficient approach to achieving fast classification of large data sets in a parallel environment, taking advantage of the capabilities of modern processors.

His divide and conquer technique combined with parallel computing achieves amazing performance for large amounts of data, and this is evident in the speedups we recorded for large sizes compared to small ones.

Thank you for your attention.

PARALLEL SYSTEMS