Report Common Assignment 2

Counting sort

Rosa Gerardo

Scovotto Luigi

Tortora Francesco

January, 2022

Index

Chapter 1

Problem description

Parallelize and Evaluate Performances of "Counting Sort" Algorithm , by using MPI.

Counting sort is an algorithm for sorting integer numbers. The computational complexity is equal to $O(n)$.

In this case study the starting point for the counting sort function was from this video: https://www.youtube.com/watch?v=qcOoEjdYSz0

1.1 Experimental setup

1.1.1 Hardware

CPU

```
machdep.cpu.brand_string: Apple M1 Max
machdep.cpu.core_count: 10
machdep.cpu.cores_per_package: 10
machdep.cpu.logical_per_package: 10
machdep.cpu.thread_count: 10

This CPU has 10 cores but 8 are for high-performance
and 2 for energetical efficiency
```

RAM

```
kern.ipc.mb_memory_pressure_percentage: 80
```

```
kern.dtrace.buffer_memory_inuse: 0
kern.dtrace.buffer_memory_maxsize: 11453246122
kern.memorystatus_apps_idle_delay_time: 10
kern.memorystatus_level: 92
kern.memorystatus_sysprocs_idle_delay_time: 10
kern.memorystatus_purge_on_critical: 8
kern.memorystatus_purge_on_urgent: 5
kern.memorystatus_purge_on_warning: 2
vm.memory_pressure: 0
hw.memsize: 34359738368
hw.optional.ucnormal_mem: 1
audit.session.member_clear_sflags_mask: 16384
audit.session.member_set_sflags_mask: 0
unified memory: 32 GB
type: LPDDR5
```

1.1.2 Software

- macOS Monterey Version 12.1
- clang version 13.0.0
- The swap is done dynamically, it is 1024MB by default, but as soon as this threshold is reached it is increased to 2048MB and so on.

Chapter 2

Performance, Speedup & Efficiency

In the first case study all the elements to be sorted are divided equally between the processes but the master process only handles the elements given by the rest of the division (the left elements).

In the second, however, all elements are divided equally, including the father. So the father performs both the number of elements that the other processes perform and the left elements.

2.1 Case study n°1

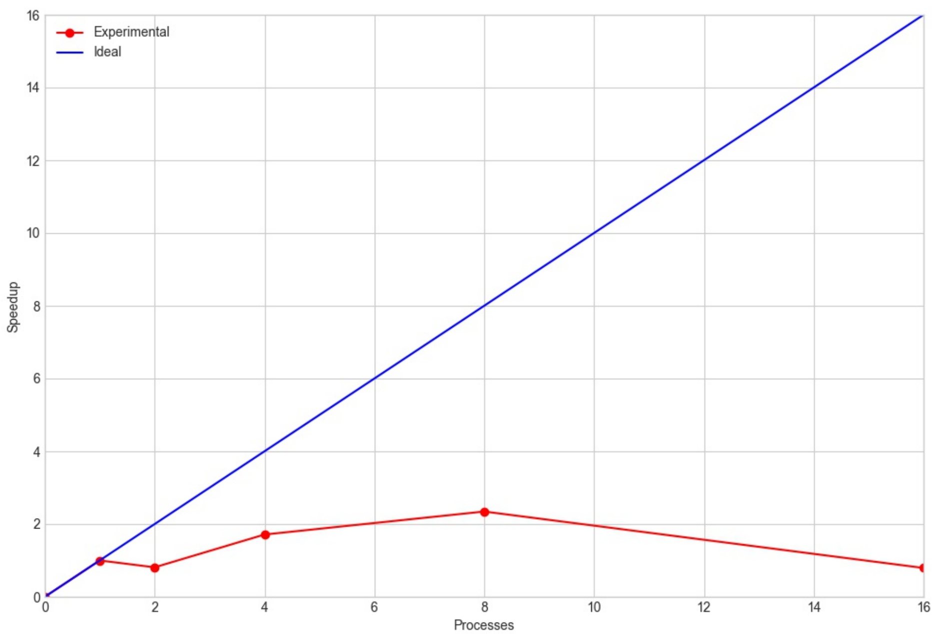
In this case study, the main purpose was to analyze the performance of program in the following build setup:

- The parallel programs are compiled with the gcc optimization -Ox where x = 1,2,3

So here we want to highlight the difference between parallel program run with different number of processes, compiled with the compiler optimizations. Furthermore the case study is done on multiple size that are 50000000, 100000000, 150000000, 200000000, with different number of Processes (1, 2, 4, 8, 16) on 50 repetitions.

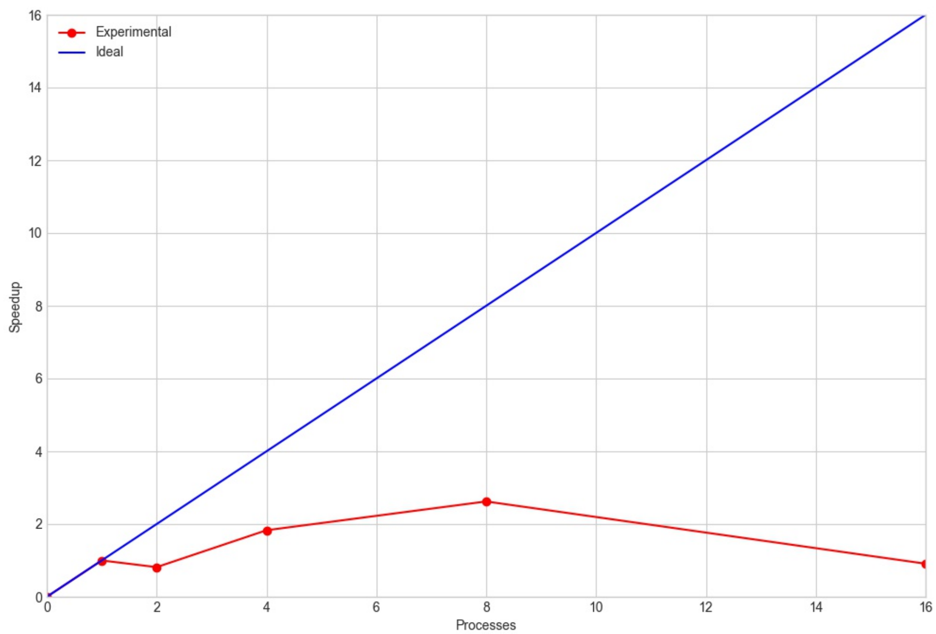
2.1.1 Size-50000000-O1

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	0,356 12	0,111 29	0,467 41	1,000 00	1,000 00
Parallel	2	0,435 48	0,141 13	0,576 61	0,810 61	0,405 30
Parallel	4	0,170 21	0,102 78	0,272 99	1,712 15	0,428 04
Parallel	8	0,103 60	0,095 75	0,199 36	2,344 57	0,293 07
Parallel	16	0,312 93	0,276 42	0,589 35	0,793 09	0,049 57



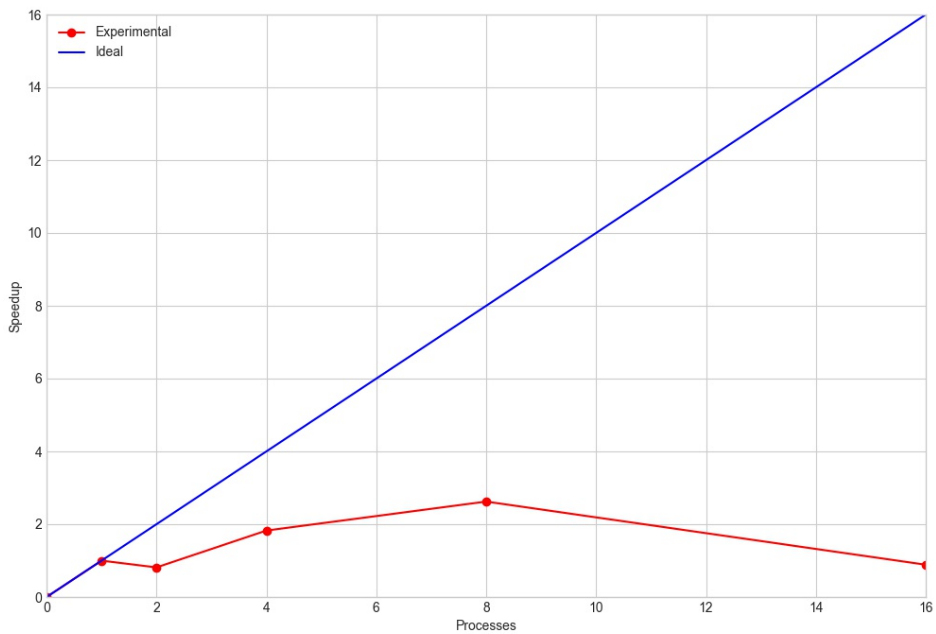
2.1.2 Size-50000000-O2

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	0,353 70	0,084 63	0,438 33	1,000 00	1,000 00
Parallel	2	0,427 24	0,110 29	0,537 52	0,815 47	0,407 73
Parallel	4	0,160 50	0,079 04	0,239 54	1,829 85	0,457 46
Parallel	8	0,096 11	0,071 08	0,167 19	2,621 82	0,327 73
Parallel	16	0,285 65	0,195 87	0,481 53	0,910 29	0,056 89



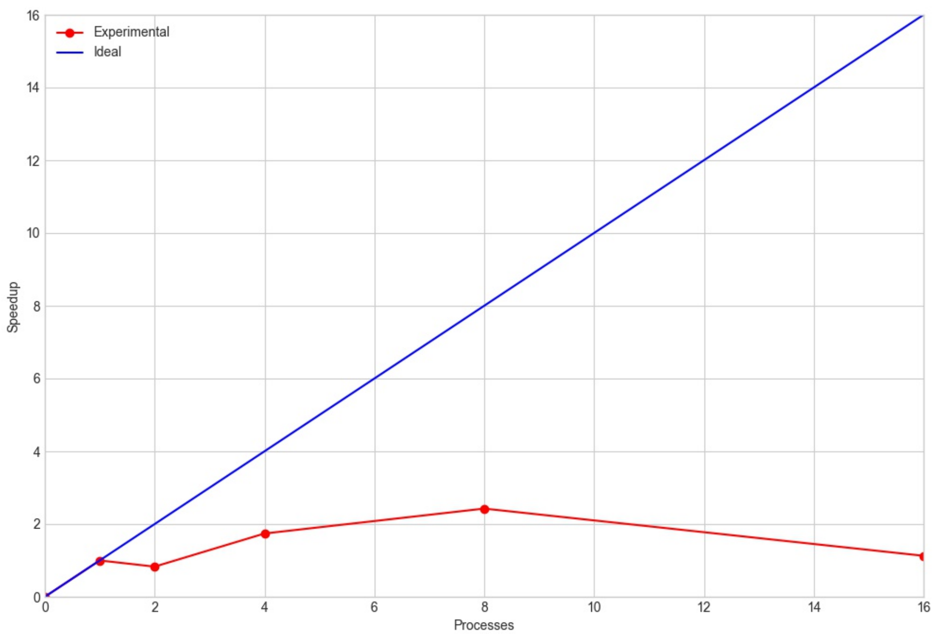
2.1.3 Size-50000000-O3

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	0,353 41	0,084 21	0,437 62	1,000 00	1,000 00
Parallel	2	0,426 91	0,110 68	0,537 60	0,814 03	0,407 02
Parallel	4	0,160 49	0,079 18	0,239 66	1,825 97	0,456 49
Parallel	8	0,096 05	0,070 84	0,166 89	2,622 17	0,327 77
Parallel	16	0,294 55	0,198 49	0,493 05	0,887 58	0,055 47



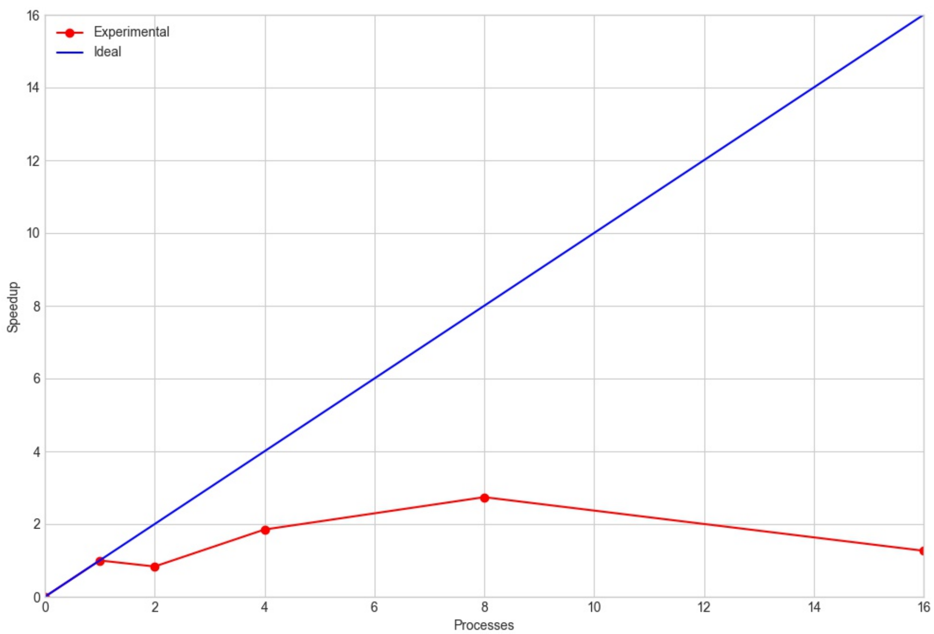
2.1.4 Size-100000000-O1

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	0,709 18	0,221 74	0,930 92	1,000 00	1,000 00
Parallel	2	0,855 98	0,266 15	1,122 13	0,829 60	0,414 80
Parallel	4	0,327 26	0,207 67	0,534 93	1,740 26	0,435 06
Parallel	8	0,191 79	0,192 05	0,383 84	2,425 28	0,303 16
Parallel	16	0,427 42	0,398 07	0,825 49	1,127 73	0,070 48



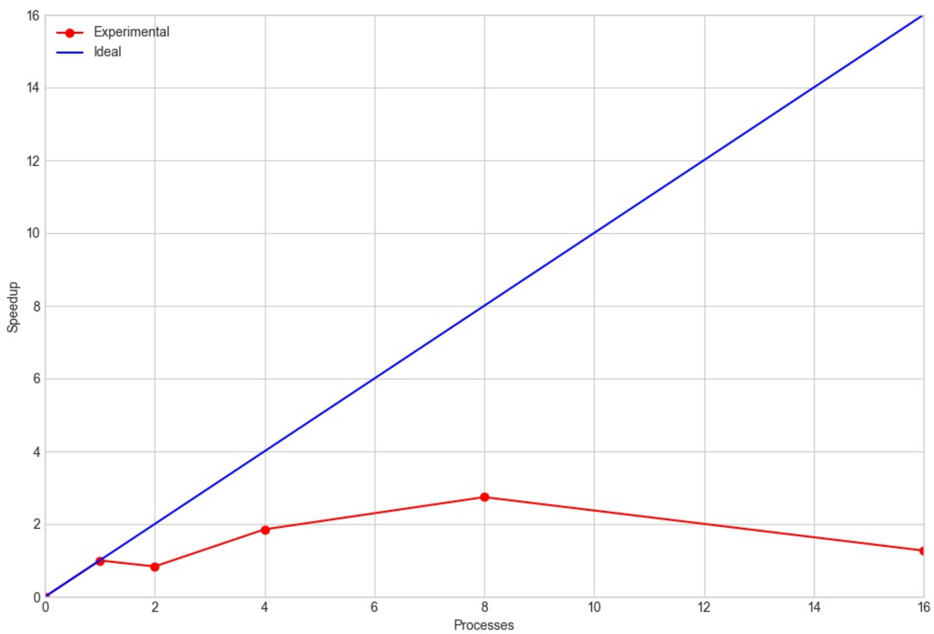
2.1.5 Size-100000000-O2

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	0,706 37	0,167 03	0,873 41	1,000 00	1,000 00
Parallel	2	0,837 91	0,210 06	1,047 97	0,833 43	0,416 71
Parallel	4	0,313 00	0,159 23	0,472 24	1,849 50	0,462 38
Parallel	8	0,175 69	0,143 13	0,318 82	2,739 48	0,342 44
Parallel	16	0,398 76	0,290 84	0,689 60	1,266 55	0,079 16



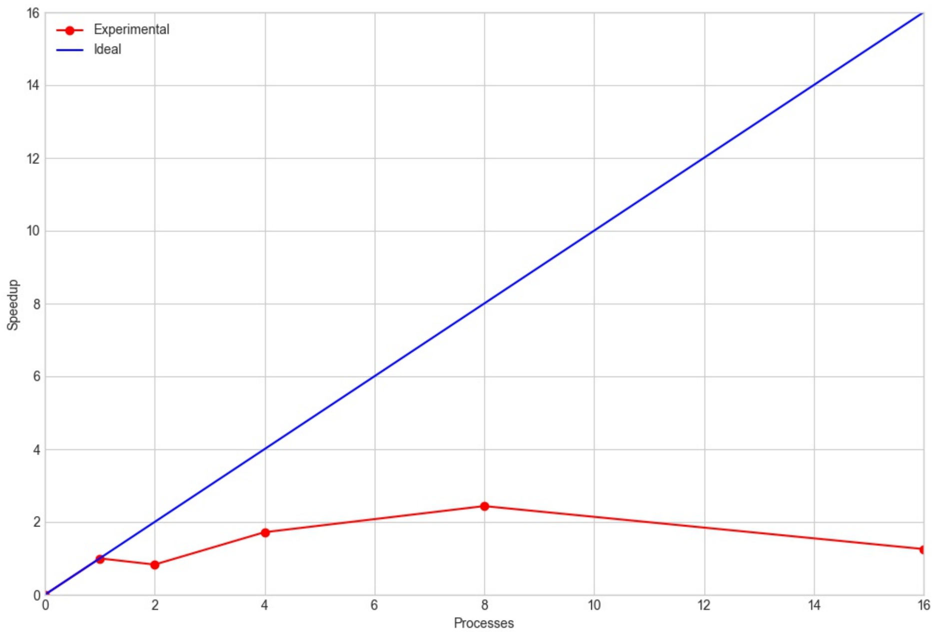
2.1.6 Size-100000000-O3

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	0,707 77	0,167 45	0,875 22	1,000 00	1,000 00
Parallel	2	0,839 09	0,209 67	1,048 76	0,834 53	0,417 26
Parallel	4	0,313 12	0,159 00	0,472 11	1,853 83	0,463 46
Parallel	8	0,177 00	0,142 37	0,319 37	2,740 46	0,342 56
Parallel	16	0,389 79	0,298 83	0,688 63	1,270 97	0,079 44



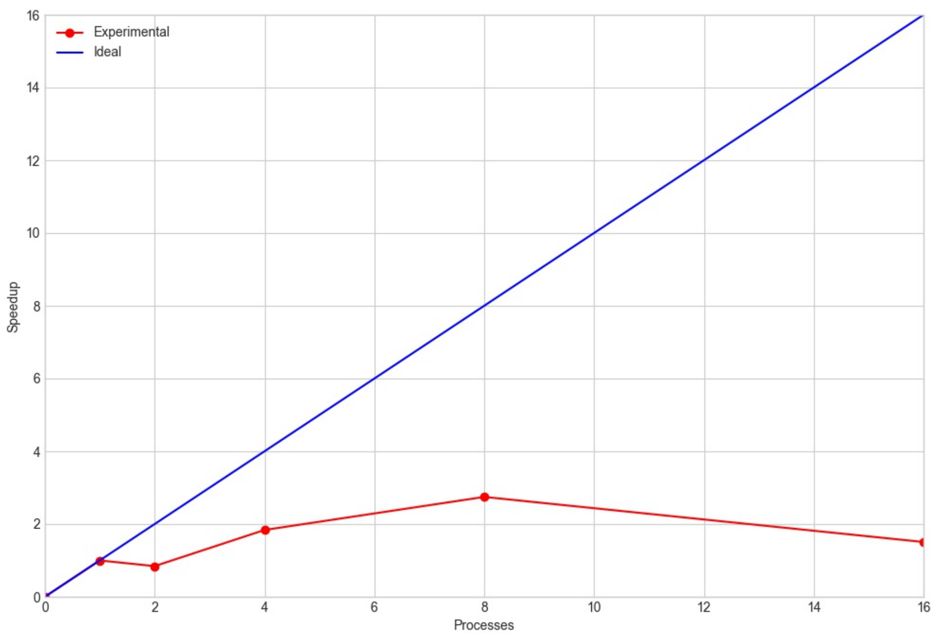
2.1.7 Size-150000000-O1

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	1,062 07	0,332 52	1,394 59	1,000 00	1,000 00
Parallel	2	1,281 68	0,398 00	1,679 67	0,830 27	0,415 14
Parallel	4	0,503 72	0,307 73	0,811 45	1,718 65	0,429 66
Parallel	8	0,285 13	0,287 33	0,572 46	2,436 12	0,304 51
Parallel	16	0,547 89	0,562 24	1,110 13	1,256 24	0,078 51



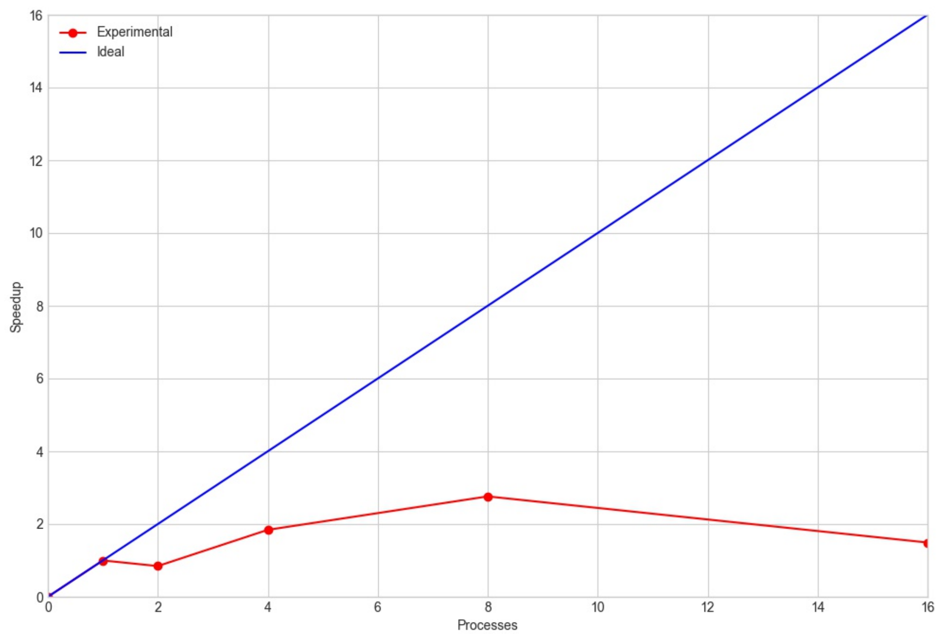
2.1.8 Size-150000000-O2

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	1,061 84	0,250 31	1,312 15	1,000 00	1,000 00
Parallel	2	1,249 89	0,306 33	1,556 22	0,843 17	0,421 58
Parallel	4	0,475 15	0,239 04	0,714 20	1,837 24	0,459 31
Parallel	8	0,265 28	0,212 58	0,477 86	2,745 90	0,343 24
Parallel	16	0,492 69	0,378 08	0,870 77	1,506 88	0,094 18



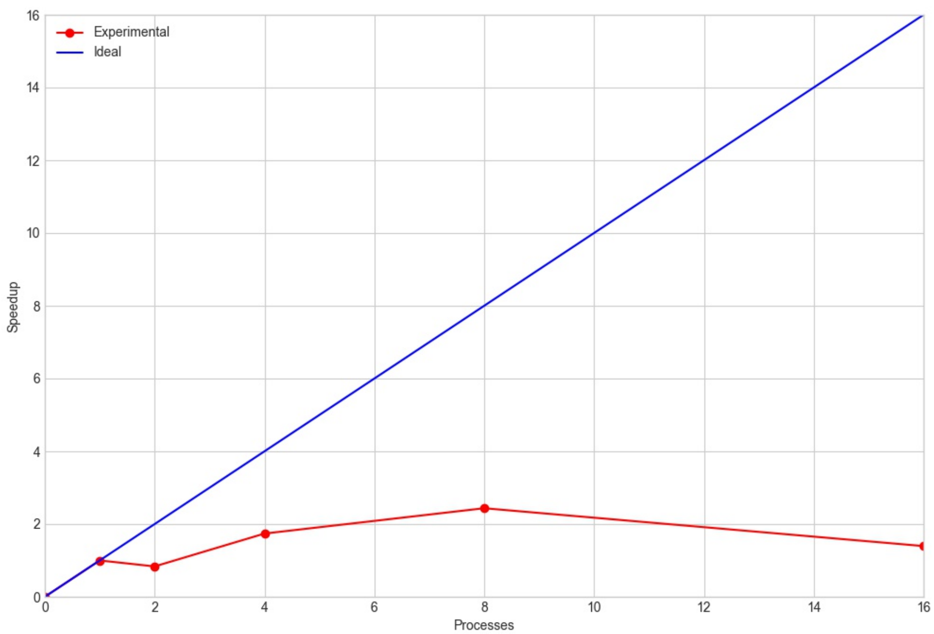
2.1.9 Size-150000000-O3

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	1,062 09	0,251 63	1,313 72	1,000 00	1,000 00
Parallel	2	1,249 50	0,305 01	1,554 51	0,845 10	0,422 55
Parallel	4	0,474 60	0,239 01	0,713 61	1,840 95	0,460 24
Parallel	8	0,263 50	0,212 46	0,475 96	2,760 13	0,345 02
Parallel	16	0,503 86	0,376 44	0,880 30	1,492 35	0,093 27



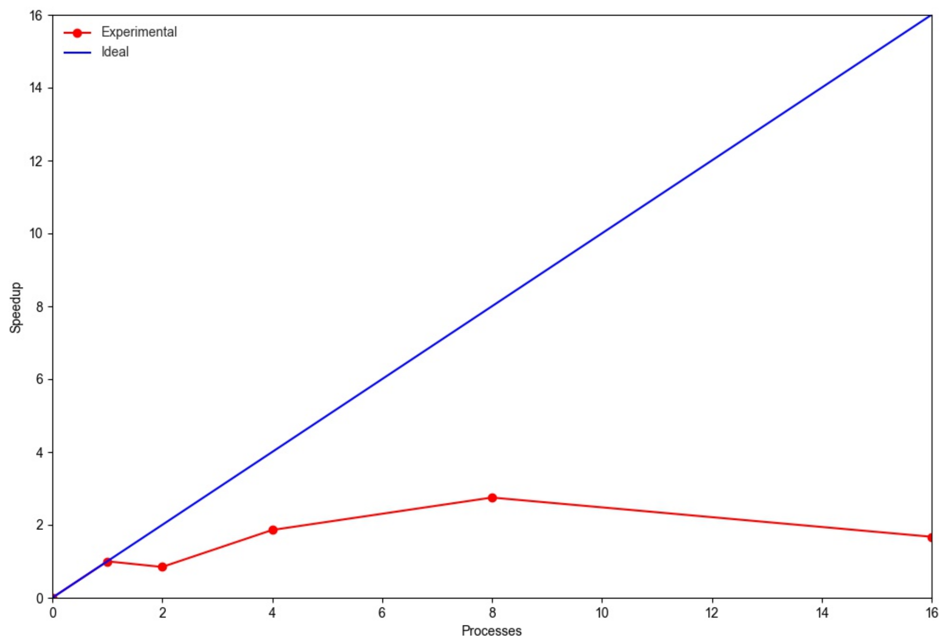
2.1.10 Size-200000000-O1

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	1,415 66	0,440 98	1,856 64	1,000 00	1,000 00
Parallel	2	1,694 25	0,528 00	2,222 25	0,835 48	0,417 74
Parallel	4	0,652 60	0,415 13	1,067 73	1,738 86	0,434 72
Parallel	8	0,375 17	0,387 29	0,762 46	2,435 08	0,304 38
Parallel	16	0,664 36	0,666 79	1,331 15	1,394 76	0,087 17



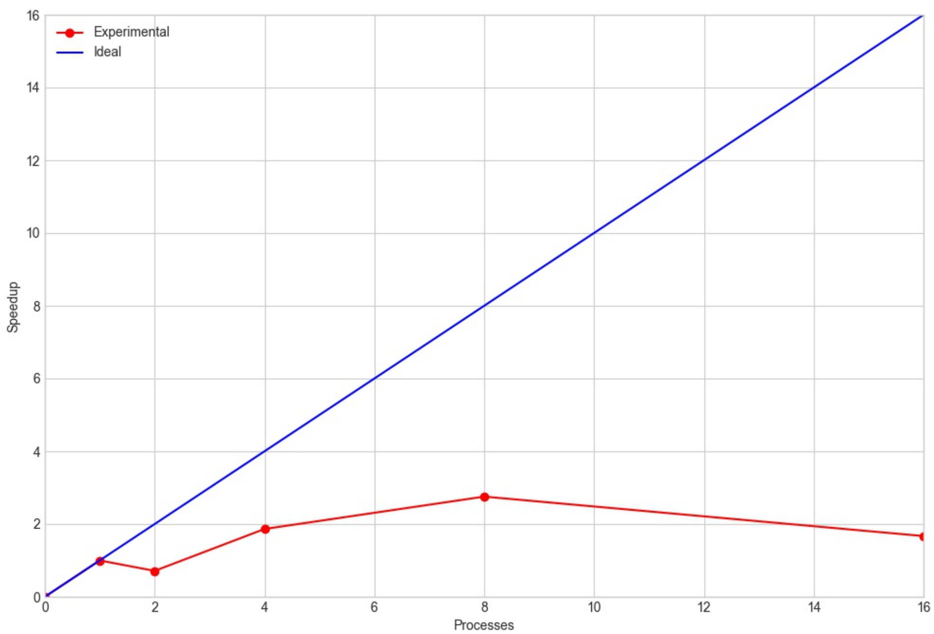
2.1.11 Size-200000000-O2

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	1,415 62	0,333 19	1,748 81	1,000 00	1,000 00
Parallel	2	1,655 52	0,412 61	2,068 13	0,845 60	0,422 80
Parallel	4	0,617 60	0,322 97	0,940 58	1,859 29	0,464 82
Parallel	8	0,345 03	0,290 91	0,635 94	2,749 95	0,343 74
Parallel	16	0,604 49	0,440 40	1,044 90	1,673 67	0,104 60



2.1.12 Size-200000000-O3

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	1,417 38	0,333 91	1,751 29	1,000 00	1,000 00
Parallel	2	2,044 88	0,411 98	2,456 86	0,712 82	0,356 41
Parallel	4	0,617 50	0,321 99	0,939 49	1,864 08	0,466 02
Parallel	8	0,345 93	0,289 83	0,635 76	2,754 62	0,344 33
Parallel	16	0,607 79	0,440 75	1,048 54	1,670 23	0,104 39



2.2 Case study n°2

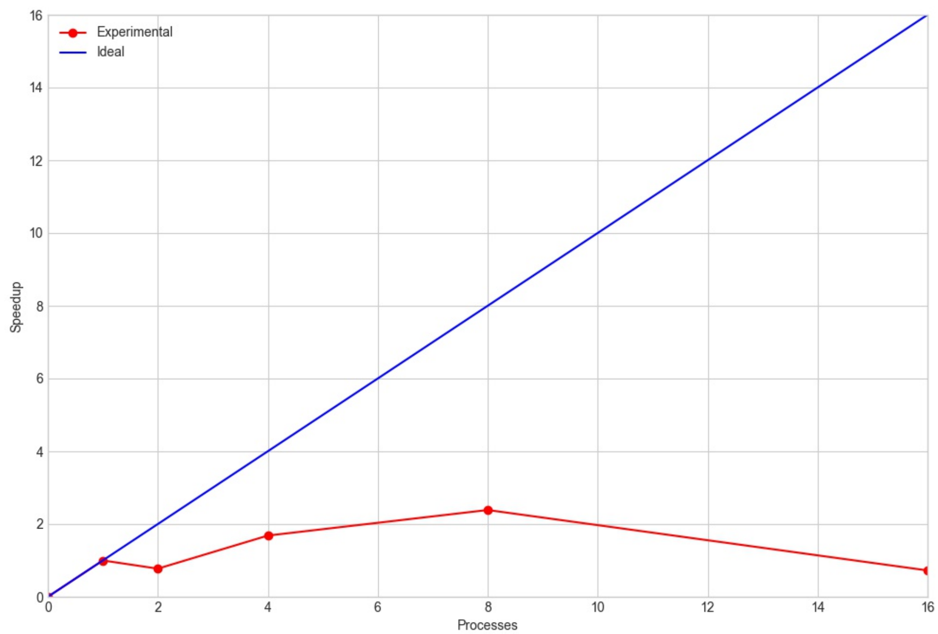
In this case study, the main purpose was to analyze the performance of program in the following build setup:

- The parallel programs are compiled with the gcc optimization -Ox where x = 1,2,3

So here we want to highlight the difference between parallel program run with different number of processes, compiled with the compiler optimizations. Furthermore the case study is done on multiple size that are 50000000, 100000000, 150000000, 200000000, with different number of Processes (1, 2, 4, 8, 16) on 50 repetitions.

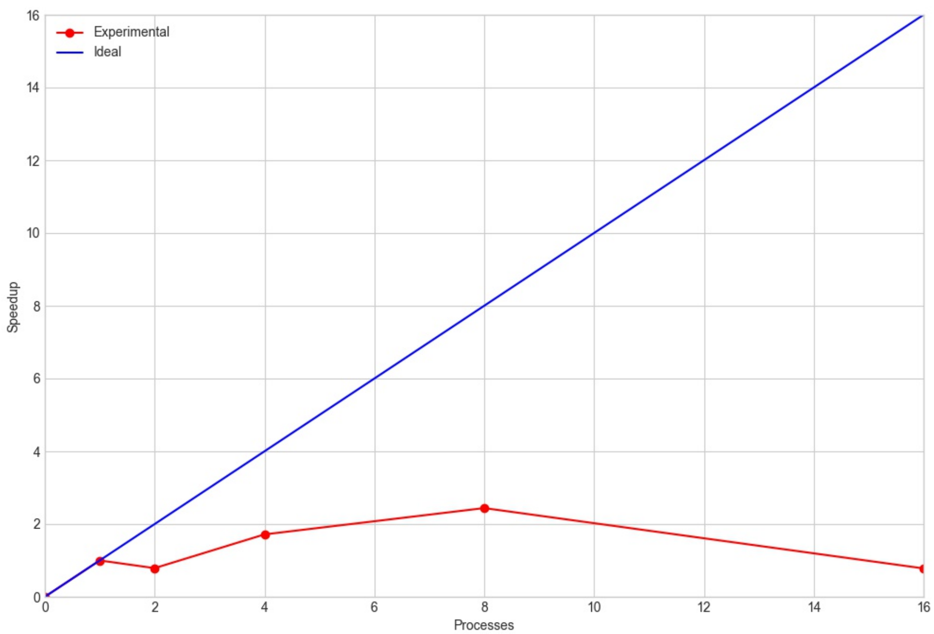
2.2.1 Size-50000000-O1

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	0,354 80	0,038 39	0,393 19	1,000 00	1,000 00
Parallel	2	0,435 45	0,072 70	0,508 15	0,773 76	0,386 88
Parallel	4	0,169 50	0,064 09	0,233 58	1,683 30	0,420 83
Parallel	8	0,103 64	0,061 14	0,164 78	2,386 14	0,298 27
Parallel	16	0,315 86	0,226 58	0,542 44	0,724 85	0,045 30



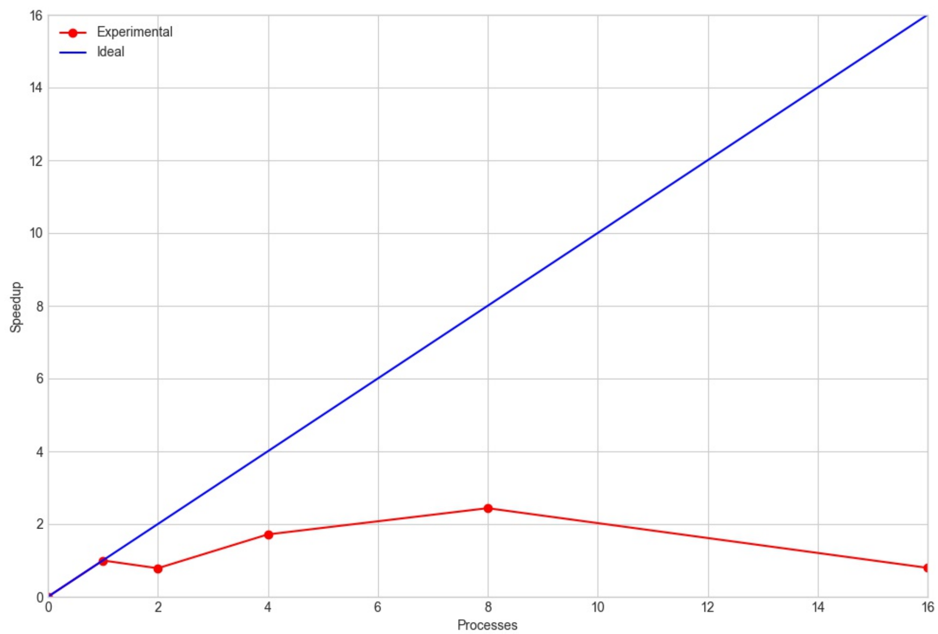
2.2.2 Size-50000000-O2

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	0,353 48	0,038 34	0,391 82	1,000 00	1,000 00
Parallel	2	0,425 93	0,071 17	0,497 10	0,788 20	0,394 10
Parallel	4	0,160 40	0,067 98	0,228 38	1,715 68	0,428 92
Parallel	8	0,095 99	0,064 65	0,160 64	2,439 08	0,304 88
Parallel	16	0,307 91	0,193 86	0,501 77	0,780 87	0,048 80



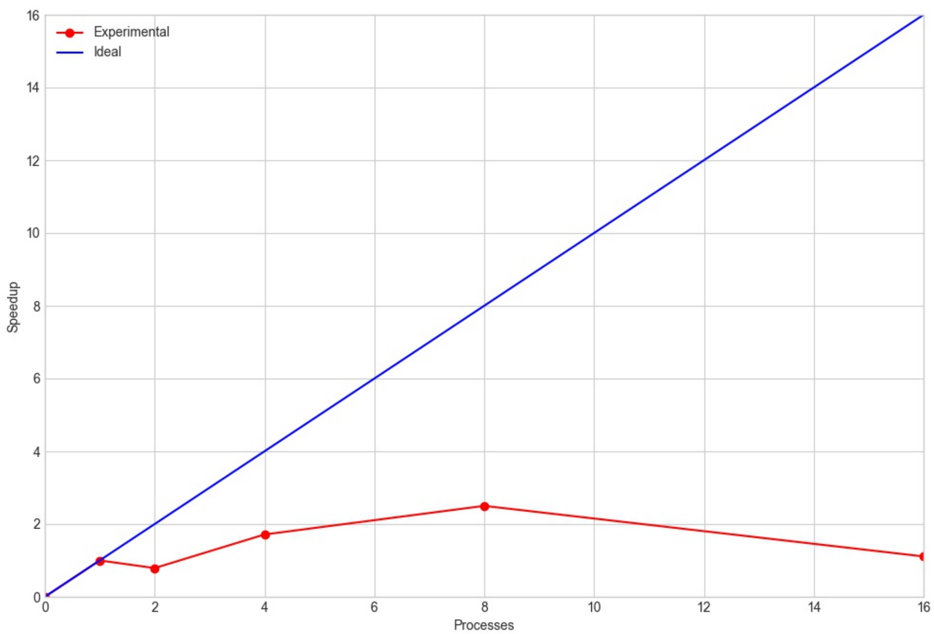
2.2.3 Size-50000000-O3

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	0,353 70	0,038 24	0,391 95	1,000 00	1,000 00
Parallel	2	0,426 97	0,072 04	0,499 01	0,785 46	0,392 73
Parallel	4	0,160 54	0,067 98	0,228 51	1,715 20	0,428 80
Parallel	8	0,096 13	0,064 73	0,160 86	2,436 60	0,304 58
Parallel	16	0,299 85	0,192 19	0,492 04	0,796 58	0,049 79



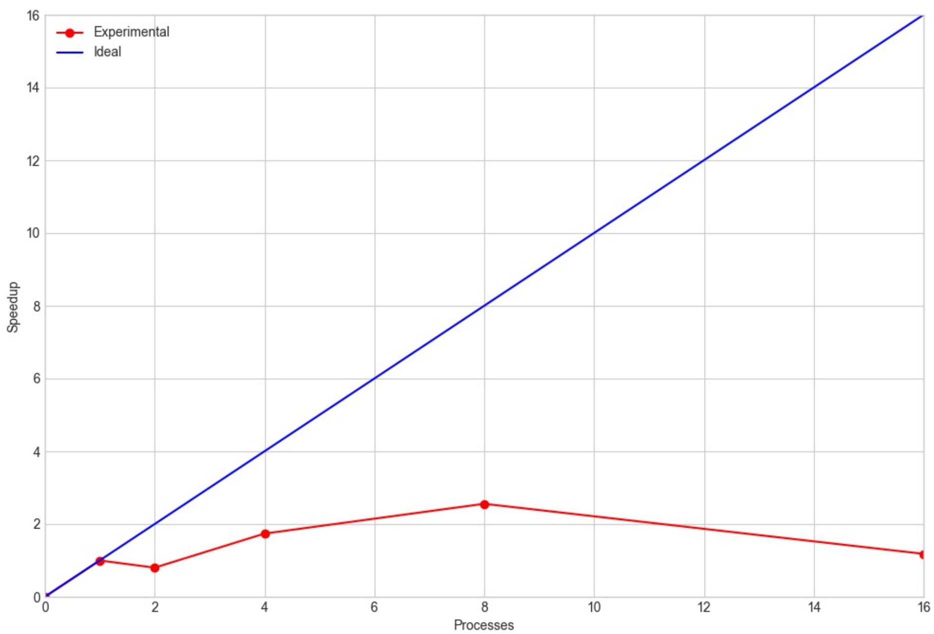
2.2.4 Size-100000000-O1

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	0,709 89	0,076 91	0,786 80	1,000 00	1,000 00
Parallel	2	0,858 59	0,137 92	0,996 51	0,789 55	0,394 78
Parallel	4	0,327 32	0,131 94	0,459 26	1,713 20	0,428 30
Parallel	8	0,190 98	0,123 74	0,314 72	2,499 98	0,312 50
Parallel	16	0,431 77	0,277 12	0,708 89	1,109 90	0,069 37



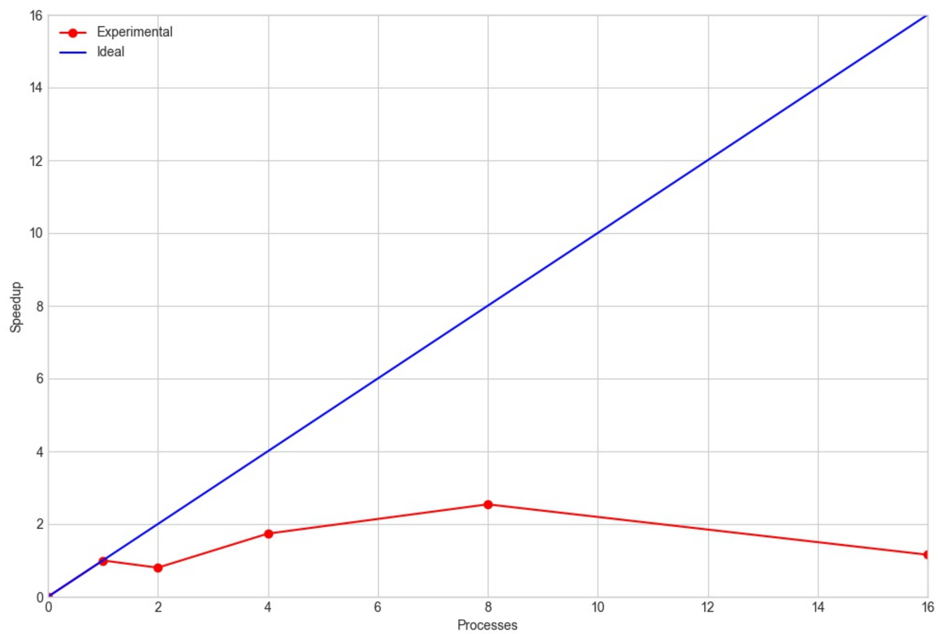
2.2.5 Size-100000000-O2

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	0,706 12	0,076 35	0,782 47	1,000 00	1,000 00
Parallel	2	0,840 74	0,136 56	0,977 30	0,800 64	0,400 32
Parallel	4	0,310 37	0,139 90	0,450 27	1,737 77	0,434 44
Parallel	8	0,176 48	0,129 51	0,305 99	2,557 16	0,319 65
Parallel	16	0,396 84	0,265 61	0,662 45	1,181 16	0,073 82



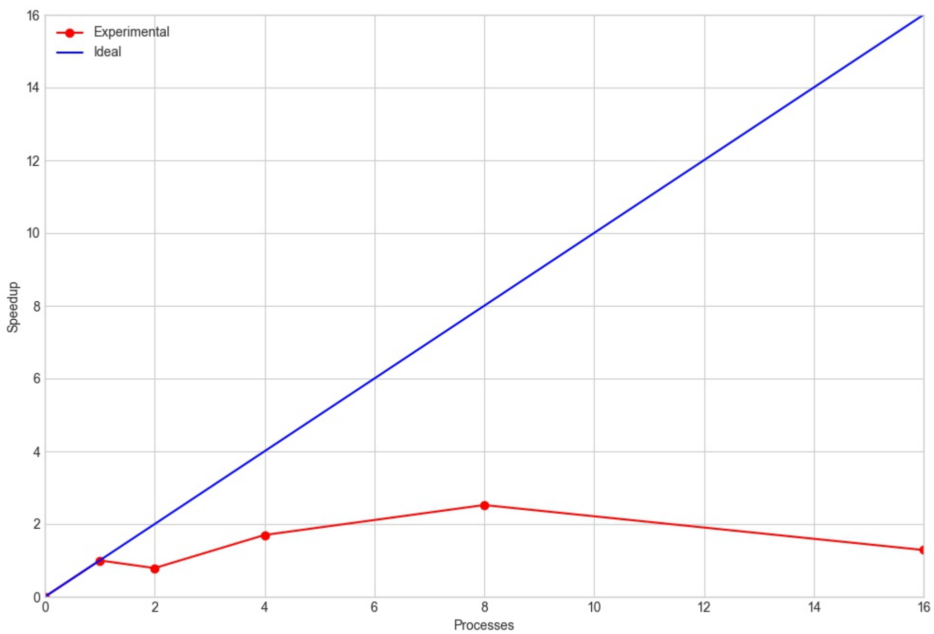
2.2.6 Size-100000000-O3

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	0,706 48	0,076 37	0,782 85	1,000 00	1,000 00
Parallel	2	0,840 57	0,136 60	0,977 18	0,801 13	0,400 57
Parallel	4	0,310 53	0,140 15	0,450 68	1,737 05	0,434 26
Parallel	8	0,178 40	0,129 53	0,307 94	2,542 26	0,317 78
Parallel	16	0,400 34	0,276 44	0,676 78	1,156 72	0,072 30



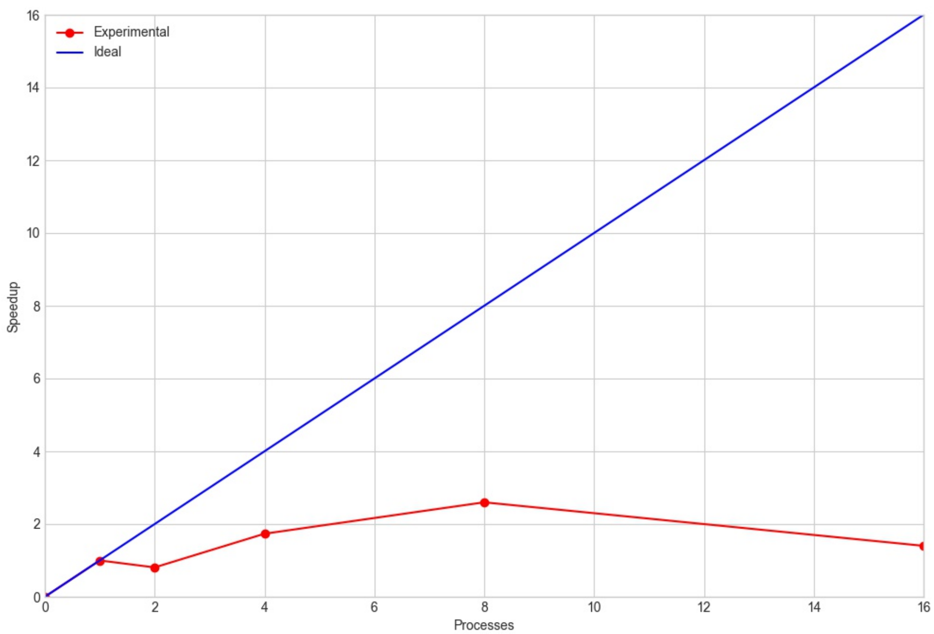
2.2.7 Size-150000000-O1

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	1,061 45	0,114 73	1,176 18	1,000 00	1,000 00
Parallel	2	1,277 17	0,212 13	1,489 31	0,789 75	0,394 88
Parallel	4	0,494 33	0,197 82	0,692 16	1,699 30	0,424 83
Parallel	8	0,279 87	0,186 28	0,466 16	2,523 15	0,315 39
Parallel	16	0,545 87	0,368 36	0,914 23	1,286 53	0,080 41



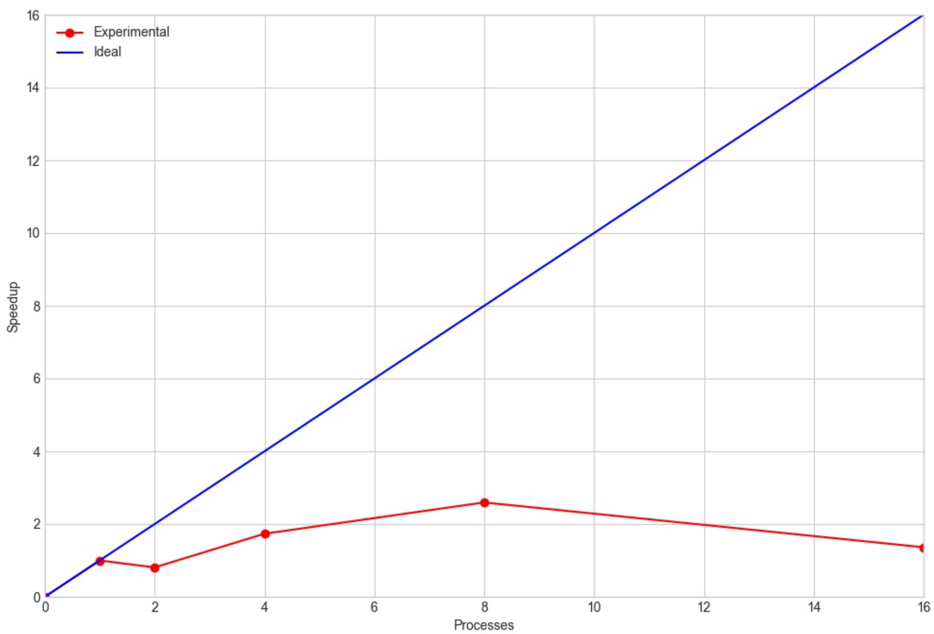
2.2.8 Size-150000000-O2

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	1,061 59	0,114 72	1,176 31	1,000 00	1,000 00
Parallel	2	1,247 40	0,208 32	1,455 71	0,808 06	0,404 03
Parallel	4	0,467 01	0,211 28	0,678 29	1,734 22	0,433 56
Parallel	8	0,257 76	0,194 95	0,452 71	2,598 39	0,324 80
Parallel	16	0,497 99	0,341 36	0,839 36	1,401 44	0,087 59



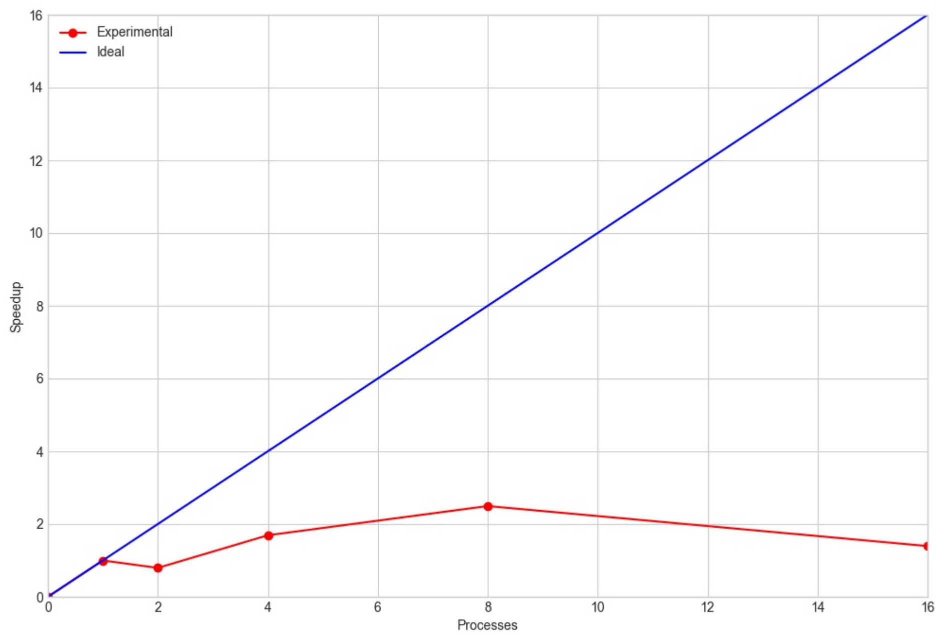
2.2.9 Size-150000000-O3

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	1,061 16	0,114 66	1,175 81	1,000 00	1,000 00
Parallel	2	1,247 99	0,208 35	1,456 35	0,807 37	0,403 69
Parallel	4	0,466 92	0,211 22	0,678 13	1,733 90	0,433 47
Parallel	8	0,258 37	0,194 95	0,453 32	2,593 76	0,324 22
Parallel	16	0,510 90	0,352 82	0,863 72	1,361 34	0,085 08



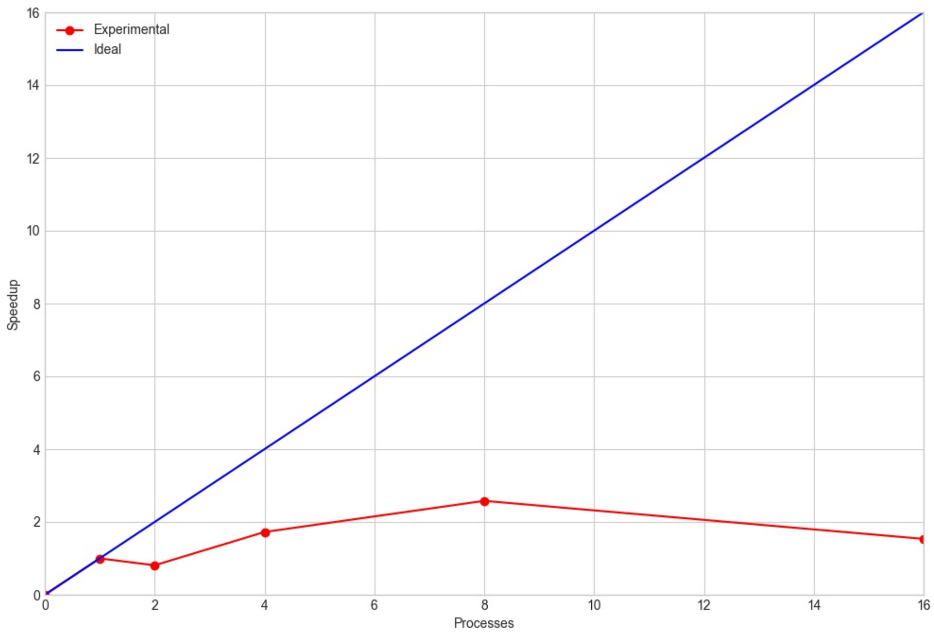
2.2.10 Size-200000000-O1

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	1,415 63	0,153 10	1,568 73	1,000 00	1,000 00
Parallel	2	1,691 69	0,282 44	1,974 14	0,794 64	0,397 32
Parallel	4	0,664 36	0,263 24	0,927 60	1,691 17	0,422 79
Parallel	8	0,381 19	0,247 59	0,628 78	2,494 87	0,311 86
Parallel	16	0,660 88	0,463 44	1,124 33	1,395 26	0,087 20



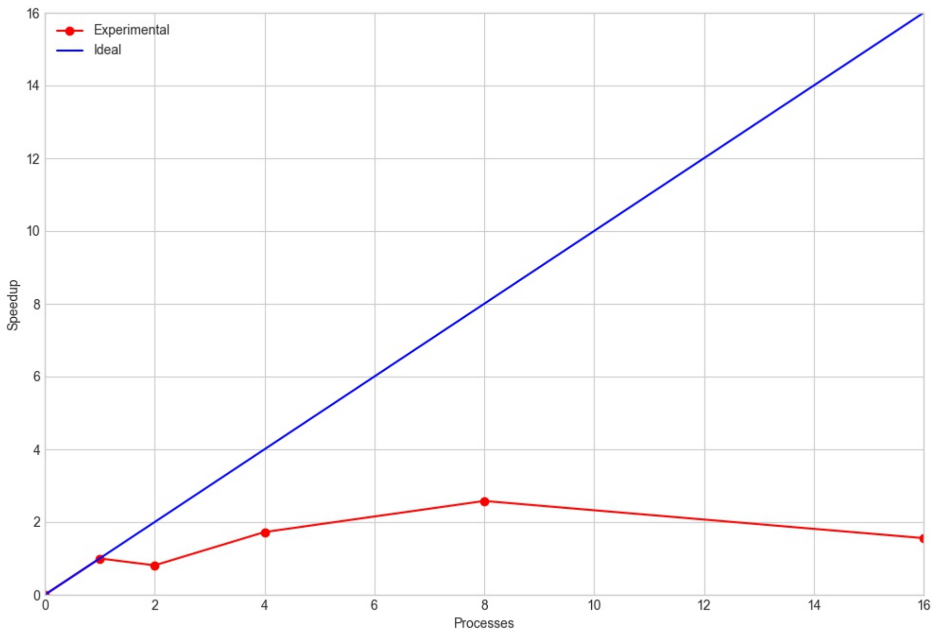
2.2.11 Size-200000000-O2

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	1,415 20	0,152 83	1,568 03	1,000 00	1,000 00
Parallel	2	1,656 23	0,279 87	1,936 10	0,809 89	0,404 95
Parallel	4	0,628 49	0,280 73	0,909 22	1,724 59	0,431 15
Parallel	8	0,348 65	0,259 11	0,607 76	2,580 00	0,322 50
Parallel	16	0,611 99	0,409 34	1,021 33	1,535 29	0,095 96



2.2.12 Size-200000000-O3

Version	Processes	Init	Counting Sort	Elapsed	Speedup	Efficiency
Parallel	1	1,415 20	0,153 17	1,568 37	1,000 00	1,000 00
Parallel	2	1,655 76	0,278 83	1,934 59	0,810 70	0,405 35
Parallel	4	0,629 09	0,280 15	0,909 24	1,724 92	0,431 23
Parallel	8	0,350 00	0,257 81	0,607 81	2,580 34	0,322 54
Parallel	16	0,597 71	0,408 34	1,006 05	1,558 94	0,097 43



Chapter 3

Considerations

3.1 Case study n°1

The maximum computed speedup is 2,8. The best configuration is with the parallel version of 8 threads on an array of 150,000,000 elements with O3 optimization.

3.2 Case study n°2

The maximum computed speedup is 2,6. The best configuration is with the parallel version of 8 threads on an array of 150,000,000 elements with O2 optimization.

3.3 Other considerations

Comparison of the two versions shows an improvement in elapsed time. The speedup of the 2 versions, however, cannot be compared because each one is calculated on the basis of the programme with its own parallel with 1 process.

We noticed that in every trial the max speedup is reached with 8 processes that is consistent with our system that has 10 physical cores and 10 threads.

The elapsed time increases steadily with the size of the problem, but the elapsed time of the parallel program doesn't increase at the same rate.

The speedup increase as processes increases, but when the program is run with 2 processes, the speedup is worse than 1; this happens because the algorithm implementation provide that all numbers are managed by child process, while the master process manage the message passing; there is a decrease of speedup also when the program is run with a processes number greater than 8 because our system has 10 cores and 10 threads.

3.4 Final considerations

We noticed that performances are influenced by different factors.

First of all we used a size of data bigger enough to overleap the overheads of the parallelism.

Then we used "MPI_Send()", instead of "MPI_Ssend()" because the first one returns to the application when the buffer is available to use, the second one always waits for the receiver to complete the Recv call.

A good load balancing is essential since it maximizes the work done in parallel. Load balancing is affected by both task distribution among processes and the set of resources that the application is running. So we did two types of balancing: in the first case, we balanced the numbers to be sorted among all child processes, the left elements due to rest of the division are managed by master process; in the second, we balanced the numbers to be sorted among all processes (master and child), also the left elements due to rest of the division are managed by master process.

Chapter 4

API

4.1 Public Functions

Type	Name
void	**generateArrayParallel**(int arr[], int n, int rank, int nprocs) This function initializes the data structure to be sorted.
int	**getMax**(int *arr, int n) Get the Max object

4.2 Public Functions Documentation

function generateArrayParallel

```
void generateArrayParallel(
    int arr[],
    int n,
    int rank,
    int nprocs
)
```

Parameters:

- *arr* a pointer to an empty array which must be populated with random numbers
- *n* size of arr
- *rank* unique number to identify process

- *nprocs* number of process

function getMax

```
int getMax(
    int *arr,
    int n
)
```

Parameters:

- *arr* array that we want calculate the maximum element
- *n* dimension of array

Returns:

- *int* 1 if is sorted, 0 if isn't.

Chapter 5

How to run

1. Create a build directory and launch cmake

```
mkdir build
cd build
cmake ..
```

2. Generate executables with

```
make
```

3. To generate measures run

```
make generate_measures
```

4. To extract mean times and speedup curves from measures run

```
make extract_measures
```

Results of measures can be found in the ‘measures/measure‘ directory divided by problem size, the gcc optimization option used and case study (V1 for case study 1 and V2 for case study 2).

You can find the complete project on GitHub: https://github.com/scov8/CommonAssignment2-Team02

The previous year's group 02 files proposed by the professor during the course were used for file generation and extraction.

The starting point for the counting sort function was from this video:
https://www.youtube.com/watch?v=qcOoEjdYSz0