First Project OMP

Performance evaluation of Counting Sort Algorithm using OMP

Briglia Maria Rosaria matr. 0622701711

Della Monica Pierpaolo matr. 0622701701

Giannino Pio Roberto matr. 0622701713

November, 2021

Index

Index

Chapter 1

Problem description

The aim of this paper is to evaluate and provide efficient solutions to the "Counting Sort" algorithm, using OpenMP library. The first provided solution was built from the pseudo-code posted on Wikipedia website, as suggested. As specified in wikipedia the time complexity of the previous algorithm is $O(n)$.
Parallelizing the algorithm it was found that some tasks can't be parallelized at all, beause of some some dependencies among datas.
- Algorithm reference: https://en.wikipedia.org/wiki/Counting_sort
The second provided solition was taken from github repository. It implements the same algorithm but with a different approach. Time complexity is $O(n^2)$ but this algorithm is completely parallelizable as it is based on nested for-loops which avoids inchoerencies among datas.
- Algorithm reference:
https://github.com/ianliu/programacao-paralela/blob/master/omp-count-sort/main.c

1.1 Experimental setup

1.1.1 Hardware

Hardware Vendor: HP

Hardware Model: HP EliteBook 850 G5

CPU

```
Architecture:        x86_64
CPU op-mode(s):      32-bit, 64-bit
Byte Order:          Little Endian
Address sizes:       39 bits physical, 48 bits virtual
CPU(s):               8
On-line CPU(s) list: 0-7
Thread(s) per core:  2
Core(s) per socket:  4
Socket(s):           1
NUMA node(s):        1
Vendor ID:           GenuineIntel
CPU family:          6
Model:               142
Model name:          Intel(R) Core(TM) i7-8550U CPU @ 1.80GHz
Stepping:            10
CPU MHz:             774.207
CPU max MHz:         4000,0000
CPU min MHz:         400,0000
BogoMIPS:            3999.93
Virtualization:      VT-x
L1d cache:           128 KiB
L1i cache:           128 KiB
L2 cache:            1 MiB
L3 cache:            8 MiB
NUMA node0 CPU(s):   0-7
```

RAM Memory

```
description: System memory
physical id: 0
size: 16GiB
description: Memory controller
product: Sunrise Point-LP PMC
vendor: Intel Corporation
physical id: 1f.2
bus info: pci@0000:00:1f.2
version: 21
width: 32 bits
clock: 33MHz (30.3ns)
configuration: latency=0
resources: memory:ba430000-ba433fff
```

1.1.2 Software

```
Operating System: Ubuntu 21.10
Kernel: Linux 5.13.0-20-generic
Architecture: x86-64
```

Chapter 2

Performance, Speedup & Efficiency

2.1 First case of study

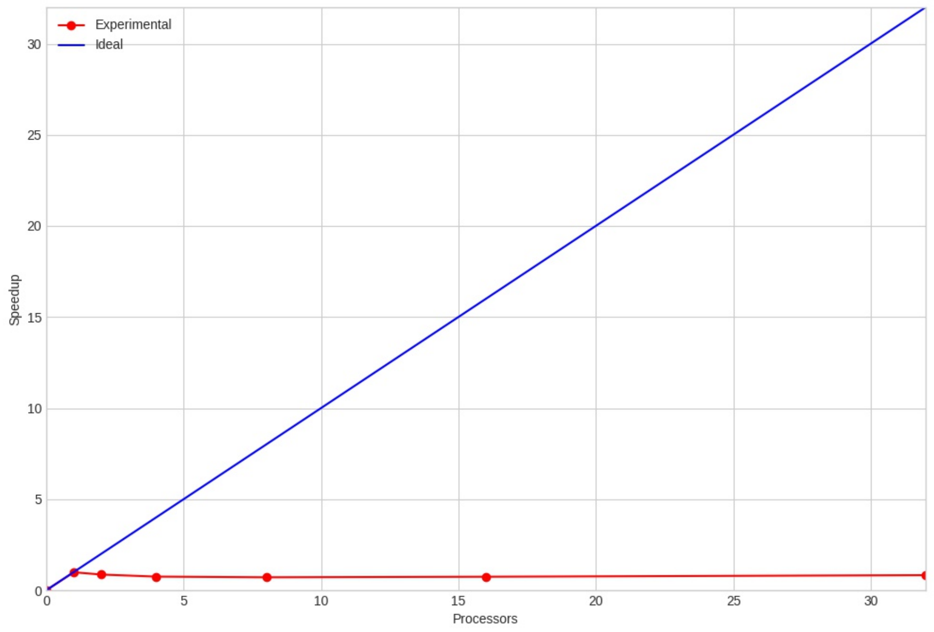
The first case of study consisted in the analysis of the algorithm with complexity $O(n)$. The analysis focused on its performance evaluation by calculating the relative efficiency and speedup. Three different setup were considered:

- The sequential program was compiled with the gcc optimization -Ox with x = 1,2,3
- The parallel programs was compiled with the gcc optimization -Ox with x = 1,2,3

The analysis is led choosing as data type **integers** due to the sturcture of the algorithm, which needs an integer bound to be executed.
The loads of the execution are **30000000** , **50000000** , **80000000** , **100000000** in order to evaluate the speedup in different load executions. Moreover it was also chosen to execute the serial version of the algorithm in order to compare the different results. For each parallel execution with the different loads, the program is ran everytime for **50** times with a different number of threads: **1, 2, 4, 8, 16, 32**. This to evaluate the trend of the speedup depending also on the thread number.

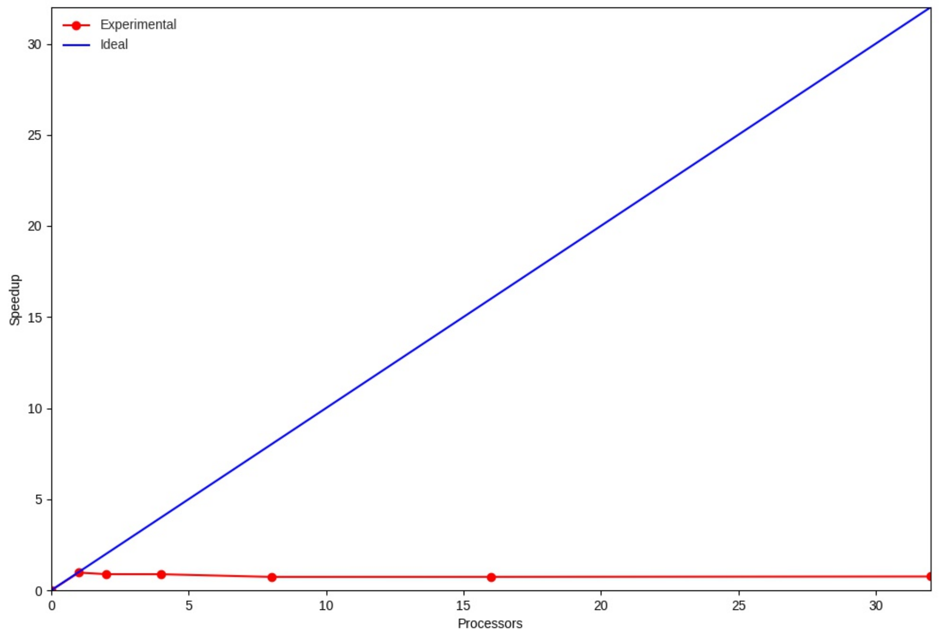
2.1.1 Size-30000000-O1

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,262 99	0,482 99	0,644 55	0,107 20	0,766 42	1,000 00	1,000 00
Parallel	1	0,263 91	0,491 21	0,665 63	0,100 25	0,775 46	0,988 34	0,988 34
Parallel	2	0,894 21	0,499 16	1,275 63	0,108 53	0,888 62	0,862 48	0,431 24
Parallel	4	2,276 36	0,578 61	2,742 33	0,109 19	1,022 97	0,749 21	0,187 30
Parallel	8	4,773 87	0,857 69	5,403 40	0,157 86	1,077 81	0,711 09	0,088 89
Parallel	16	5,697 51	0,485 80	6,014 17	0,144 62	1,033 50	0,741 58	0,046 35
Parallel	32	5,168 75	0,483 74	5,439 48	0,161 09	0,925 06	0,828 50	0,025 89



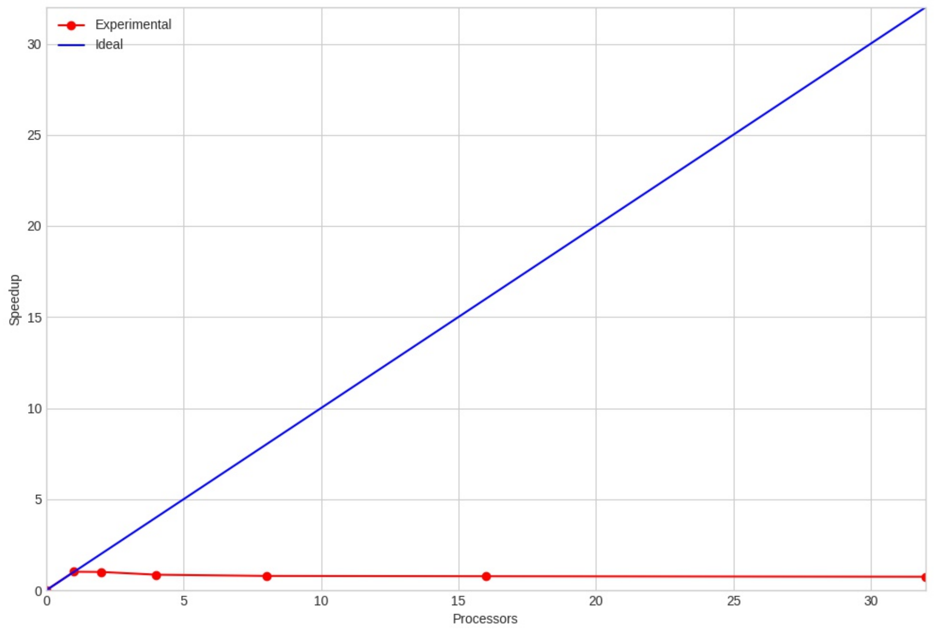
2.1.2 Size-30000000-O2

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,260 51	0,489 38	0,656 90	0,108 49	0,765 46	1,000 00	1,000 00
Parallel	1	0,269 99	0,502 72	0,677 26	0,107 60	0,790 19	0,968 70	0,968 70
Parallel	2	0,896 79	0,476 40	1,276 11	0,100 97	0,867 20	0,882 68	0,441 34
Parallel	4	2,059 86	0,512 69	2,422 35	0,110 69	0,870 23	0,879 61	0,219 90
Parallel	8	4,528 38	0,821 06	5,161 53	0,142 22	1,044 94	0,732 55	0,091 57
Parallel	16	5,771 48	0,498 74	6,049 14	0,165 83	1,042 79	0,734 05	0,045 88
Parallel	32	5,726 49	0,478 35	6,036 76	0,154 65	1,014 52	0,754 51	0,023 58



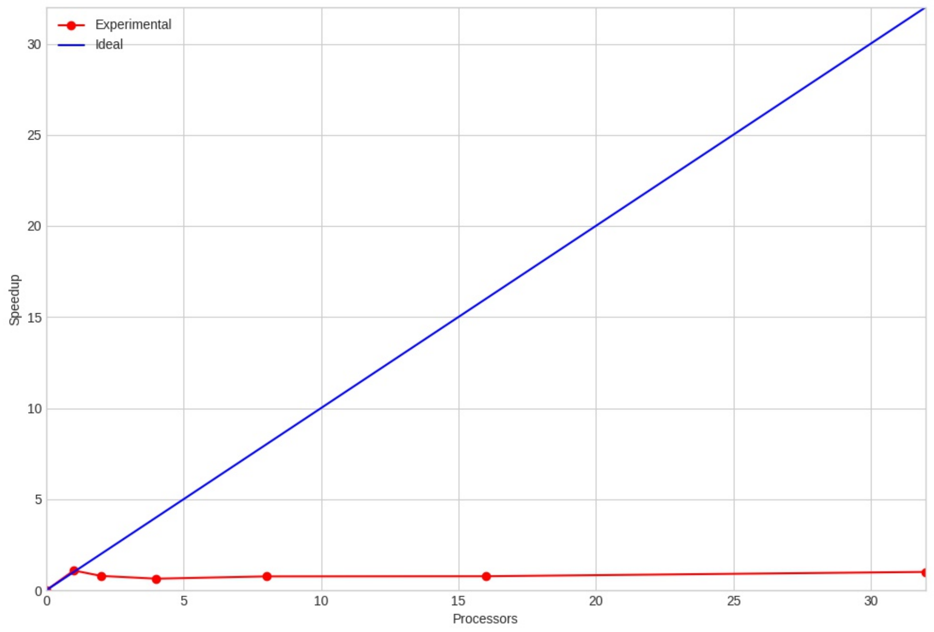
2.1.3 Size-30000000-O3

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,267 68	0,518 30	0,688 44	0,109 79	0,803 59	1,000 00	1,000 00
Parallel	1	0,263 97	0,503 72	0,670 85	0,105 89	0,791 17	1,015 71	1,015 71
Parallel	2	0,828 48	0,435 62	1,173 52	0,098 74	0,800 45	1,003 92	0,501 96
Parallel	4	2,148 35	0,562 09	2,588 50	0,113 43	0,940 91	0,854 06	0,213 52
Parallel	8	4,510 77	0,814 31	5,138 65	0,138 03	1,022 16	0,786 17	0,098 27
Parallel	16	5,693 24	0,483 20	6,033 36	0,157 47	1,043 65	0,769 99	0,048 12
Parallel	32	6,025 75	0,521 75	6,375 66	0,170 62	1,083 48	0,741 67	0,023 18



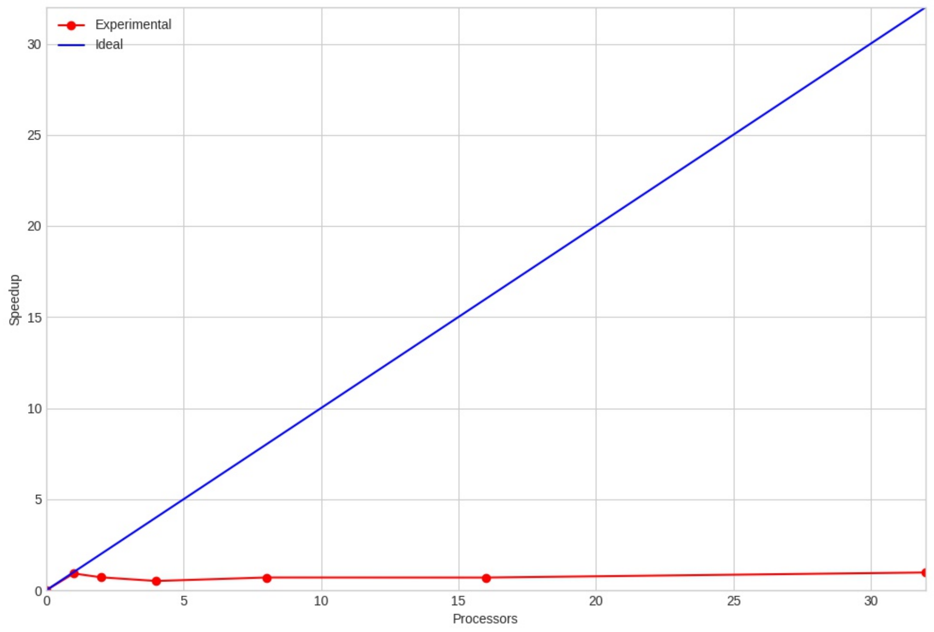
2.1.4 Size-50000000-O1

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,427 93	0,851 84	1,126 30	0,181 69	1,315 73	1,000 00	1,000 00
Parallel	1	0,401 81	0,800 90	1,048 93	0,166 22	1,212 41	1,085 22	1,085 22
Parallel	2	1,661 68	0,957 52	2,357 59	0,215 14	1,673 97	0,785 99	0,393 00
Parallel	4	4,153 71	1,250 96	5,090 74	0,269 97	2,071 39	0,635 19	0,158 80
Parallel	8	6,778 59	1,166 18	7,645 43	0,230 52	1,730 23	0,760 44	0,095 05
Parallel	16	9,405 88	0,813 84	9,972 19	0,241 36	1,710 49	0,769 21	0,048 08
Parallel	32	8,960 18	0,608 85	9,396 21	0,172 06	1,308 09	1,005 84	0,031 43



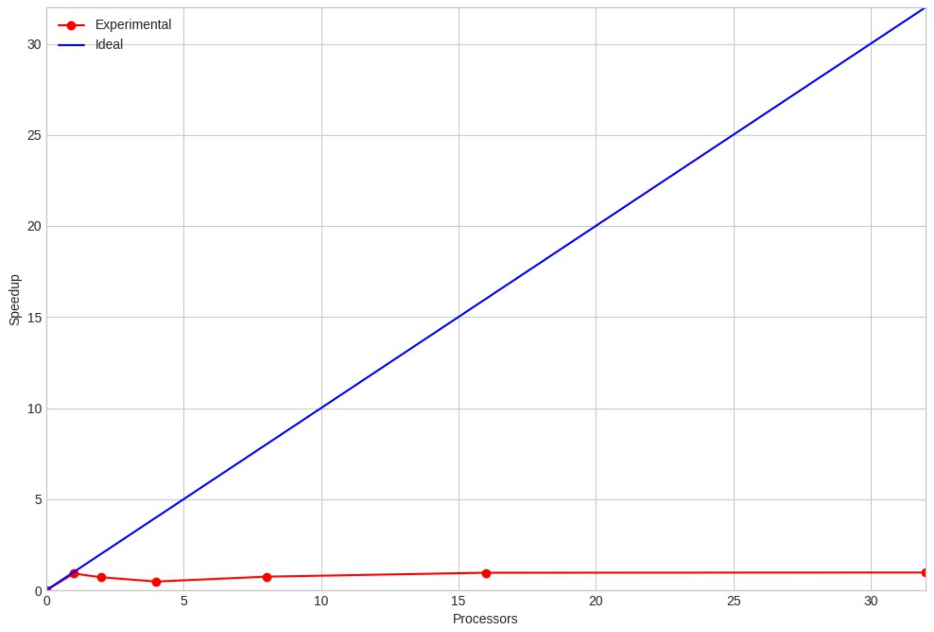
2.1.5 Size-50000000-O2

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,398 12	0,803 31	1,058 81	0,167 81	1,240 10	1,000 00	1,000 00
Parallel	1	0,432 91	0,877 30	1,169 56	0,181 81	1,347 85	0,920 06	0,920 06
Parallel	2	1,753 18	0,973 88	2,483 55	0,212 27	1,748 97	0,709 05	0,354 52
Parallel	4	3,545 95	1,347 88	4,634 20	0,308 89	2,440 76	0,508 08	0,127 02
Parallel	8	7,679 98	1,168 04	8,558 55	0,231 06	1,770 61	0,700 38	0,087 55
Parallel	16	9,703 39	0,904 73	10,375 86	0,281 15	1,778 94	0,697 10	0,043 57
Parallel	32	8,855 20	0,568 57	9,286 11	0,164 71	1,272 52	0,974 53	0,030 45



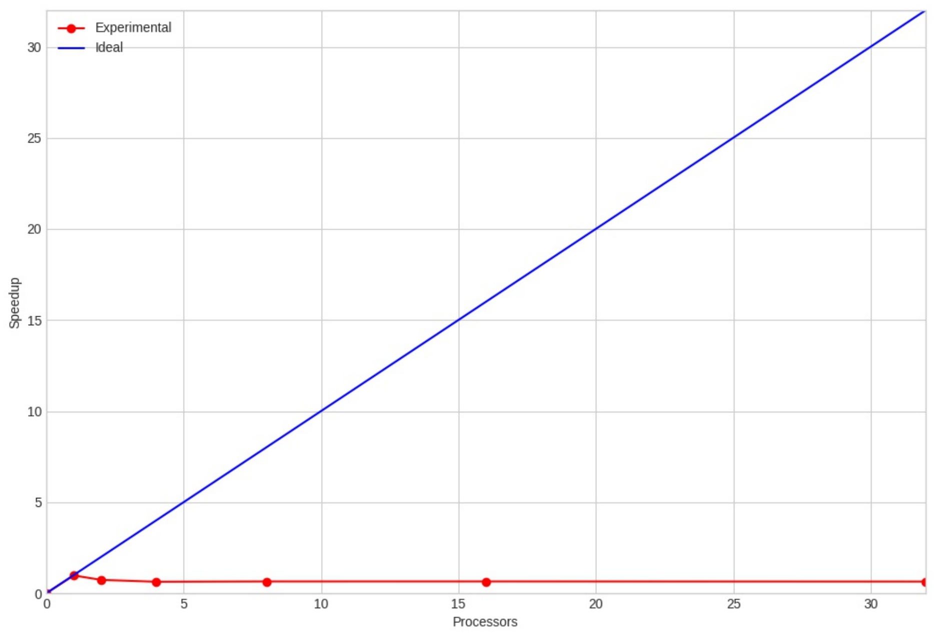
2.1.6 Size-50000000-O3

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,403 66	0,850 91	1,105 59	0,174 39	1,290 77	1,000 00	1,000 00
Parallel	1	0,471 20	0,908 91	1,197 22	0,201 03	1,408 42	0,916 46	0,916 46
Parallel	2	1,545 63	1,127 10	2,459 88	0,248 71	1,814 84	0,711 23	0,355 61
Parallel	4	2,688 18	1,292 74	3,691 17	0,294 47	2,696 97	0,478 60	0,119 65
Parallel	8	7,741 01	1,160 97	8,613 64	0,233 02	1,733 03	0,744 80	0,093 10
Parallel	16	8,695 48	0,583 55	9,149 07	0,159 42	1,344 83	0,959 80	0,059 99
Parallel	32	9,120 74	0,576 41	9,559 95	0,163 88	1,329 00	0,971 23	0,030 35



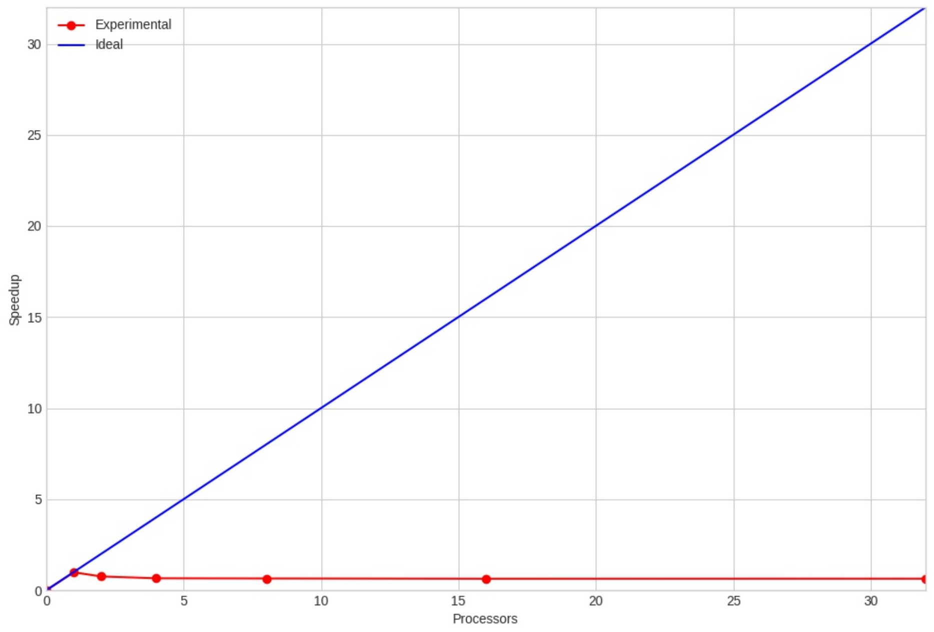
2.1.7 Size-80000000-O1

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,428 53	0,871 98	1,156 71	0,182 26	1,320 24	1,000 00	1,000 00
Parallel	1	0,421 68	0,891 10	1,175 02	0,183 03	1,352 06	0,976 47	0,976 47
Parallel	2	1,806 20	0,928 20	2,571 97	0,193 26	1,799 23	0,733 78	0,366 89
Parallel	4	4,729 02	0,939 90	5,524 09	0,208 32	2,092 97	0,630 80	0,157 70
Parallel	8	8,738 45	1,112 88	9,664 23	0,232 21	2,037 90	0,647 85	0,080 98
Parallel	16	13,163 55	0,894 04	13,827 38	0,253 33	2,034 62	0,648 89	0,040 56
Parallel	32	14,029 65	0,909 99	14,728 00	0,263 51	2,059 42	0,641 07	0,020 03



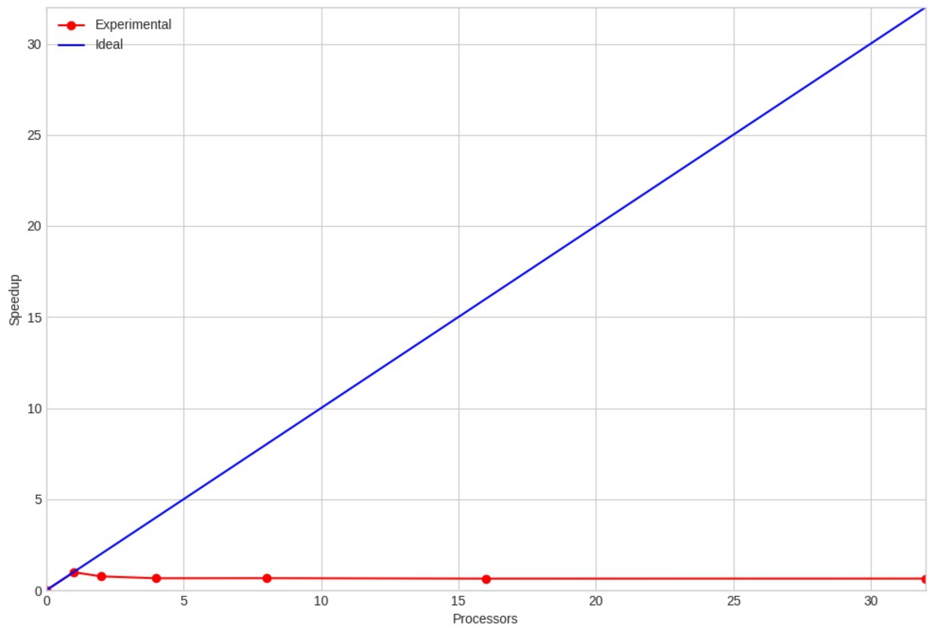
2.1.8 Size-80000000-O2

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,423 51	0,846 70	1,146 31	0,172 83	1,304 13	1,000 00	1,000 00
Parallel	1	0,424 31	0,862 76	1,142 88	0,173 80	1,324 00	0,984 99	0,984 99
Parallel	2	1,729 48	0,896 21	2,421 52	0,187 43	1,721 00	0,757 77	0,378 89
Parallel	4	4,585 58	0,916 11	5,340 00	0,209 38	1,985 03	0,656 98	0,164 25
Parallel	8	8,943 27	1,094 90	9,846 47	0,232 61	2,017 03	0,646 56	0,080 82
Parallel	16	13,572 52	0,884 33	14,236 05	0,256 51	2,065 00	0,631 54	0,039 47
Parallel	32	14,191 84	0,884 66	14,791 16	0,262 13	2,054 92	0,634 64	0,019 83



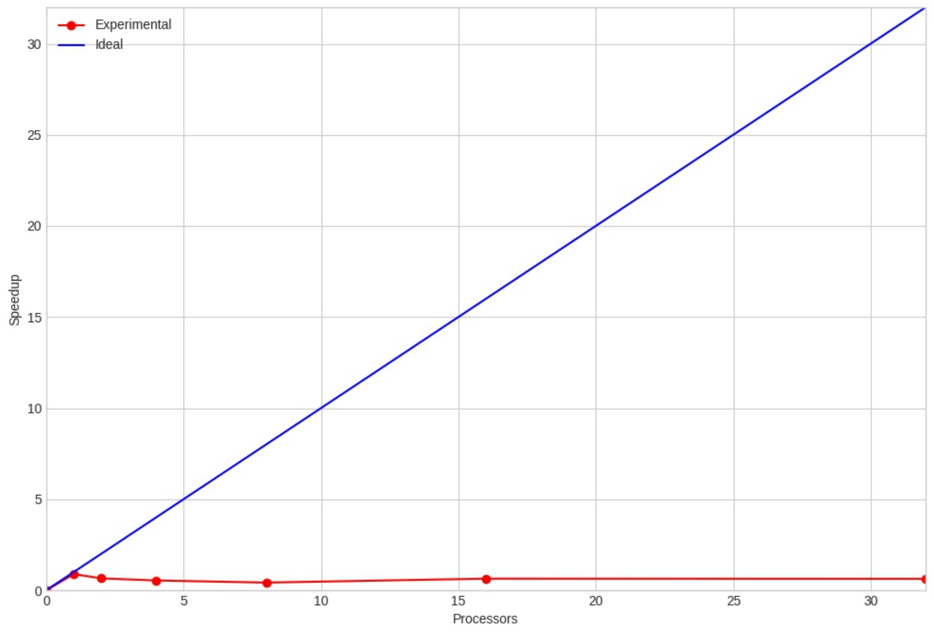
2.1.9 Size-80000000-O3

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,417 50	0,880 62	1,164 55	0,173 56	1,324 76	1,000 00	1,000 00
Parallel	1	0,424 22	0,885 29	1,162 41	0,177 13	1,338 77	0,989 53	0,989 53
Parallel	2	1,746 94	0,886 61	2,479 48	0,183 76	1,739 14	0,761 73	0,380 86
Parallel	4	4,635 59	0,920 34	5,416 97	0,208 09	2,012 03	0,658 42	0,164 60
Parallel	8	8,897 84	1,079 07	9,793 25	0,229 61	1,997 43	0,663 23	0,082 90
Parallel	16	13,509 16	0,879 46	14,179 57	0,248 43	2,075 12	0,638 40	0,039 90
Parallel	32	14,218 26	0,896 82	14,901 82	0,269 03	2,066 00	0,641 22	0,020 04



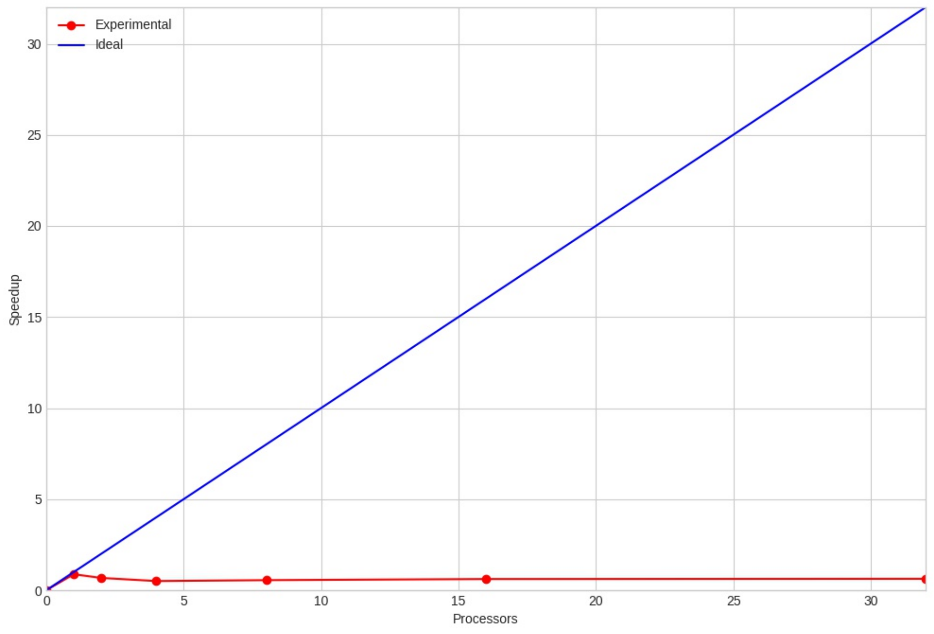
2.1.10 Size-100000000-O1

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,523 22	1,063 81	1,421 82	0,213 73	1,620 29	1,000 00	1,000 00
Parallel	1	0,583 79	1,204 50	1,578 94	0,241 93	1,829 69	0,885 56	0,885 56
Parallel	2	2,519 73	1,241 16	3,564 86	0,260 35	2,483 18	0,652 51	0,326 25
Parallel	4	6,229 31	1,528 60	7,535 73	0,315 71	3,026 72	0,535 33	0,133 83
Parallel	8	7,193 10	1,996 12	8,829 63	0,418 97	3,859 07	0,419 87	0,052 48
Parallel	16	16,484 23	1,128 18	17,334 22	0,312 03	2,550 43	0,635 30	0,039 71
Parallel	32	17,399 68	1,143 05	18,205 11	0,323 94	2,586 31	0,626 49	0,019 58



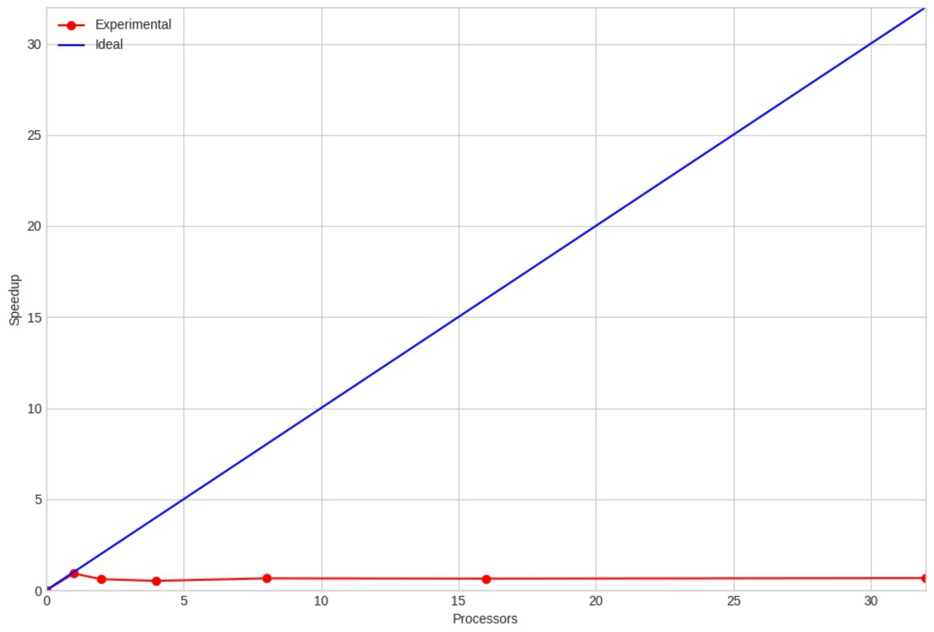
2.1.11 Size-100000000-O2

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,521 00	1,067 56	1,405 74	0,223 58	1,619 26	1,000 00	1,000 00
Parallel	1	0,588 55	1,214 06	1,603 56	0,249 20	1,843 24	0,878 49	0,878 49
Parallel	2	2,390 17	1,203 48	3,370 94	0,260 03	2,397 00	0,675 54	0,337 77
Parallel	4	6,667 93	1,602 20	8,026 17	0,354 33	3,205 85	0,505 10	0,126 27
Parallel	8	8,185 52	1,501 91	9,579 71	0,321 62	2,924 19	0,553 75	0,069 22
Parallel	16	17,272 04	1,105 60	18,115 98	0,313 33	2,631 29	0,615 39	0,038 46
Parallel	32	17,845 25	1,112 76	18,687 91	0,313 79	2,579 74	0,627 68	0,019 62



2.1.12 Size-100000000-O3

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,520 17	1,120 80	1,468 38	0,219 89	1,675 86	1,000 00	1,000 00
Parallel	1	0,586 58	1,191 34	1,593 79	0,243 23	1,826 44	0,917 55	0,917 55
Parallel	2	2,729 03	1,414 66	3,906 64	0,299 53	2,755 24	0,608 24	0,304 12
Parallel	4	5,976 07	1,665 74	7,316 00	0,382 47	3,260 06	0,514 06	0,128 51
Parallel	8	11,380 35	1,327 08	12,464 97	0,287 49	2,562 31	0,654 04	0,081 76
Parallel	16	17,190 75	1,120 41	18,042 97	0,323 72	2,629 31	0,637 38	0,039 84
Parallel	32	17,876 27	1,111 99	18,710 31	0,326 79	2,513 33	0,666 79	0,020 84



2.2 Second case of study

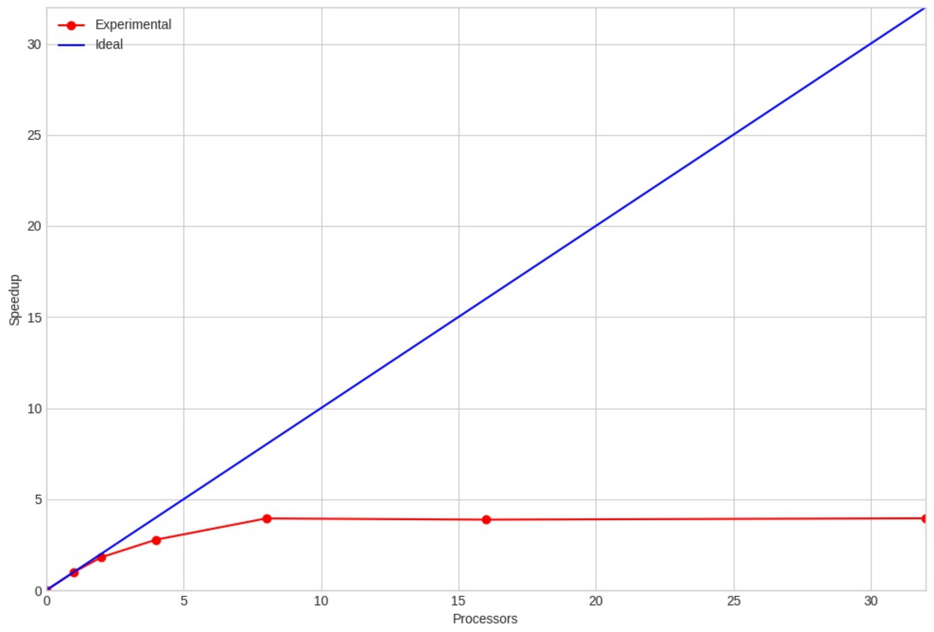
The second proposed algorithm runs the "Counting Sort" algorithm in $O(n^2)$ in its serial execution. The tests were led following the current setup:

- The sequential program was compiled with the gcc optimization -Ox with x = 1,2,3
- The parallel program was compiled with the gcc optimization -Ox with x = 1,2,3

The analysis is led choosing as data type **integers** due to the sturcture of the algorithm, which needs an integer bound to be executed.
The loads of the execution were **30000**, **50000**, **80000**, **100000** in order to evaluate the speedup in different load executions. Moreover it was also chosen to execute the serial version of the algorithm in order to compare the different results. For each parallel execution with the different loads, the program is ran everytime for **50** times with a different number of threads: **1**, **2**, **4**, **8**, **16**, **32**. This to evaluate the trend of the speedup depending also on the thread number.

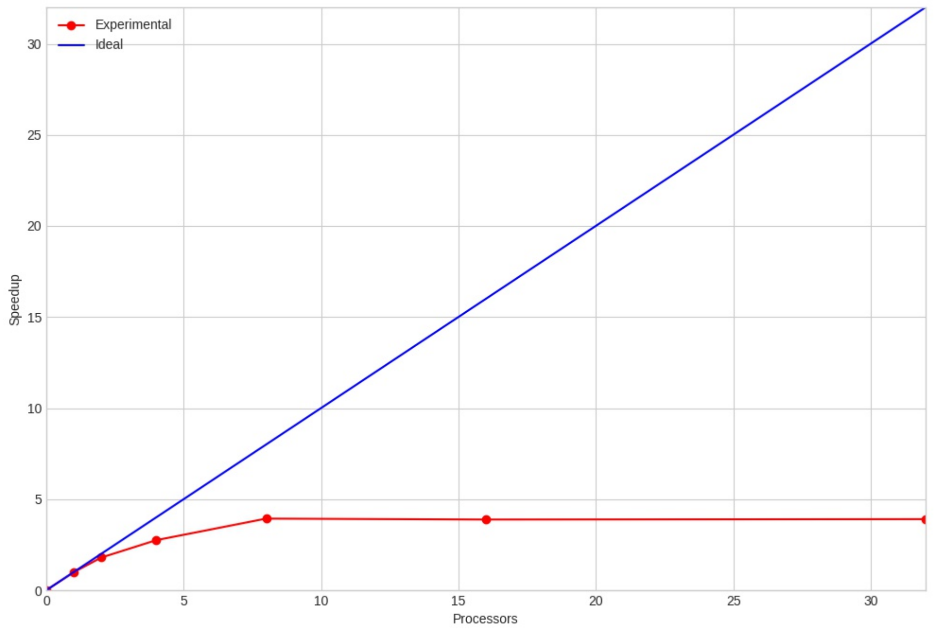
2.2.1 Size-30000-O1

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,000 13	2,070 83	2,070 38	0,000 00	2,071 87	1,000 00	1,000 00
Parallel	1	0,000 13	2,090 85	2,091 35	0,000 00	2,091 56	0,990 59	0,990 59
Parallel	2	0,000 42	2,285 53	2,283 91	0,000 00	1,144 82	1,809 77	0,904 89
Parallel	4	0,000 48	2,961 57	2,966 23	0,000 00	0,746 26	2,776 32	0,694 08
Parallel	8	0,001 00	4,156 69	4,167 85	0,000 35	0,525 53	3,942 45	0,492 81
Parallel	16	0,003 99	4,037 32	4,040 28	0,001 33	0,534 73	3,874 61	0,242 16
Parallel	32	0,004 38	4,061 09	4,063 19	0,002 36	0,524 61	3,949 34	0,123 42



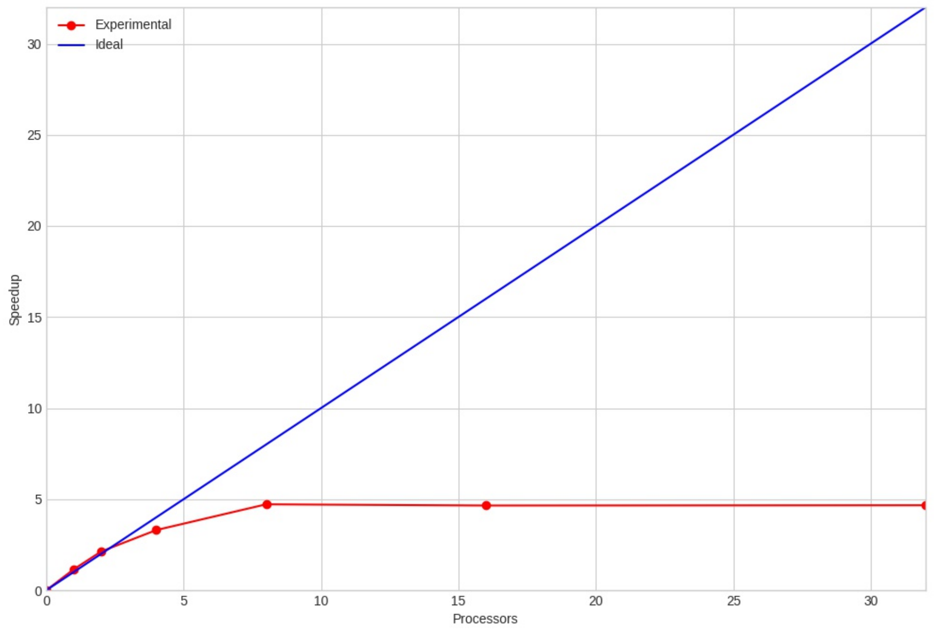
2.2.2 Size-30000-O2

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,000 13	2,022 05	2,021 87	0,000 00	2,023 26	1,000 00	1,000 00
Parallel	1	0,000 13	2,039 79	2,040 54	0,000 00	2,040 82	0,991 40	0,991 40
Parallel	2	0,000 43	2,235 15	2,235 81	0,000 00	1,127 70	1,794 16	0,897 08
Parallel	4	0,000 53	2,889 96	2,892 33	0,000 00	0,736 11	2,748 57	0,687 14
Parallel	8	0,000 94	4,076 18	4,094 21	0,001 11	0,514 55	3,932 09	0,491 51
Parallel	16	0,004 01	3,983 85	3,986 78	0,000 98	0,521 36	3,880 74	0,242 55
Parallel	32	0,004 57	4,001 59	4,005 07	0,002 27	0,518 37	3,903 14	0,121 97



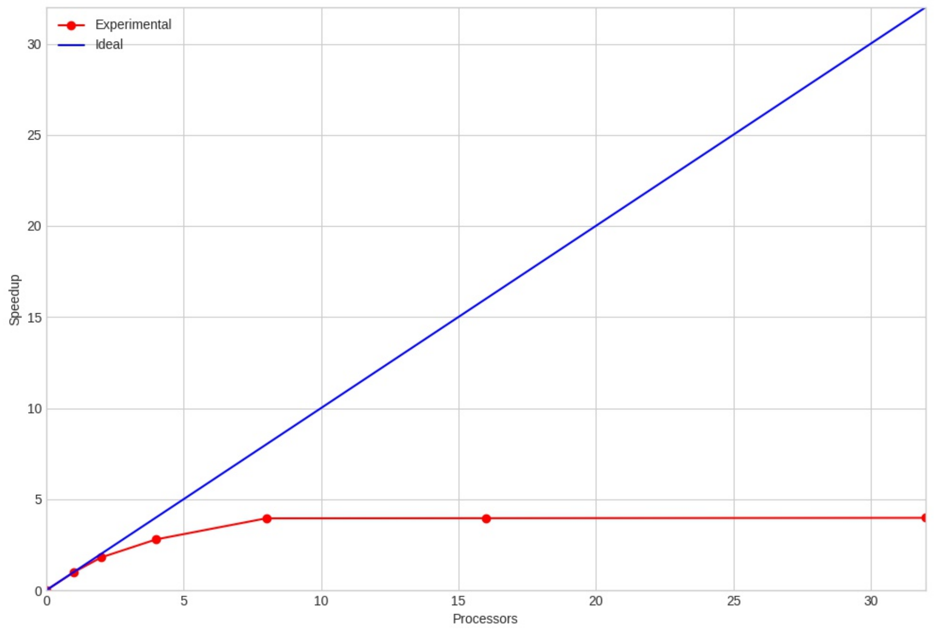
2.2.3 Size-30000-O3

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,000 13	2,446 80	2,447 67	0,000 00	2,447 78	1,000 00	1,000 00
Parallel	1	0,000 13	2,128 16	2,128 97	0,000 00	2,129 10	1,149 68	1,149 68
Parallel	2	0,000 42	2,294 68	2,296 96	0,000 00	1,149 42	2,129 57	1,064 78
Parallel	4	0,000 45	2,938 04	2,941 16	0,000 00	0,741 32	3,301 90	0,825 47
Parallel	8	0,000 79	4,110 33	4,130 14	0,000 00	0,518 90	4,717 27	0,589 66
Parallel	16	0,003 89	4,014 87	4,017 85	0,000 00	0,526 29	4,651 04	0,290 69
Parallel	32	0,004 40	4,063 75	4,067 02	0,000 98	0,524 65	4,665 55	0,145 80



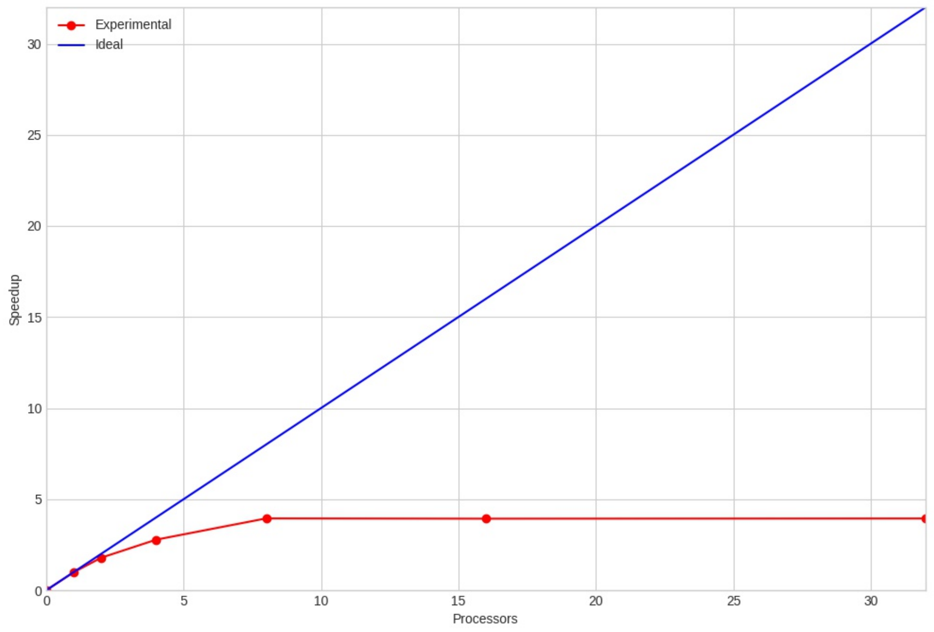
2.2.4 Size-50000-O1

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,000 26	5,732 24	5,732 66	0,000 00	5,733 59	1,000 00	1,000 00
Parallel	1	0,000 26	5,799 84	5,800 71	0,000 00	5,800 75	0,988 42	0,988 42
Parallel	2	0,000 58	6,355 46	6,357 91	0,000 00	3,180 91	1,802 50	0,901 25
Parallel	4	0,000 78	8,185 89	8,188 24	0,000 00	2,053 49	2,792 12	0,698 03
Parallel	8	0,001 90	11,505 54	11,515 18	0,000 00	1,452 91	3,946 27	0,493 28
Parallel	16	0,006 26	11,360 89	11,363 53	0,000 98	1,449 88	3,954 52	0,247 16
Parallel	32	0,006 64	11,378 44	11,373 51	0,003 07	1,442 68	3,974 28	0,124 20



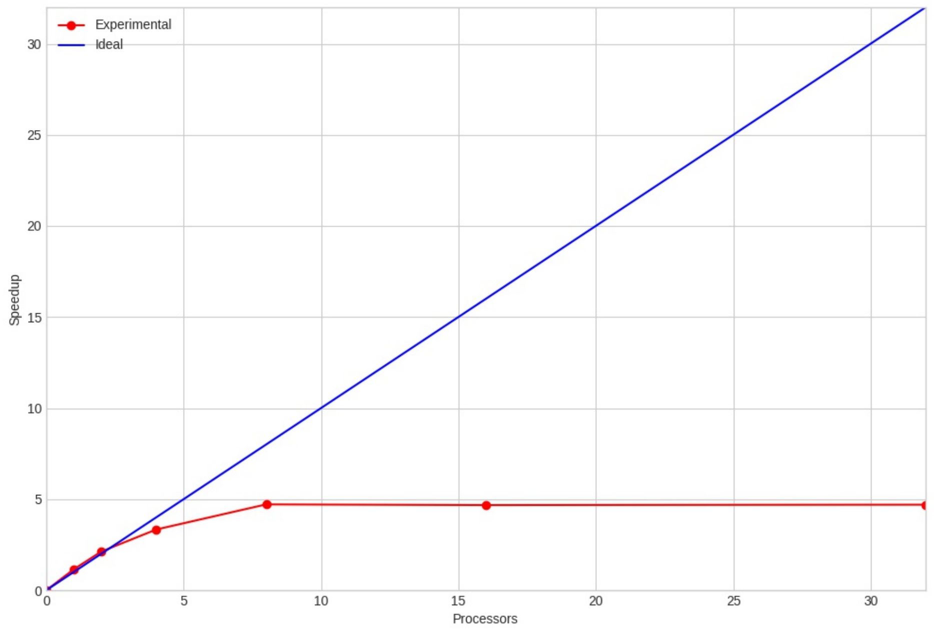
2.2.5 Size-50000-O2

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,000 26	5,609 48	5,610 23	0,000 00	5,610 47	1,000 00	1,000 00
Parallel	1	0,000 26	5,657 24	5,657 81	0,000 00	5,658 53	0,991 51	0,991 51
Parallel	2	0,000 67	6,207 20	6,208 49	0,000 00	3,140 59	1,786 44	0,893 22
Parallel	4	0,000 67	7,935 30	7,936 71	0,000 00	2,020 69	2,776 51	0,694 13
Parallel	8	0,001 37	11,311 70	11,318 37	0,001 42	1,423 08	3,942 48	0,492 81
Parallel	16	0,005 99	11,178 01	11,181 14	0,002 13	1,428 03	3,928 82	0,245 55
Parallel	32	0,006 59	11,225 51	11,230 05	0,002 33	1,424 38	3,938 90	0,123 09



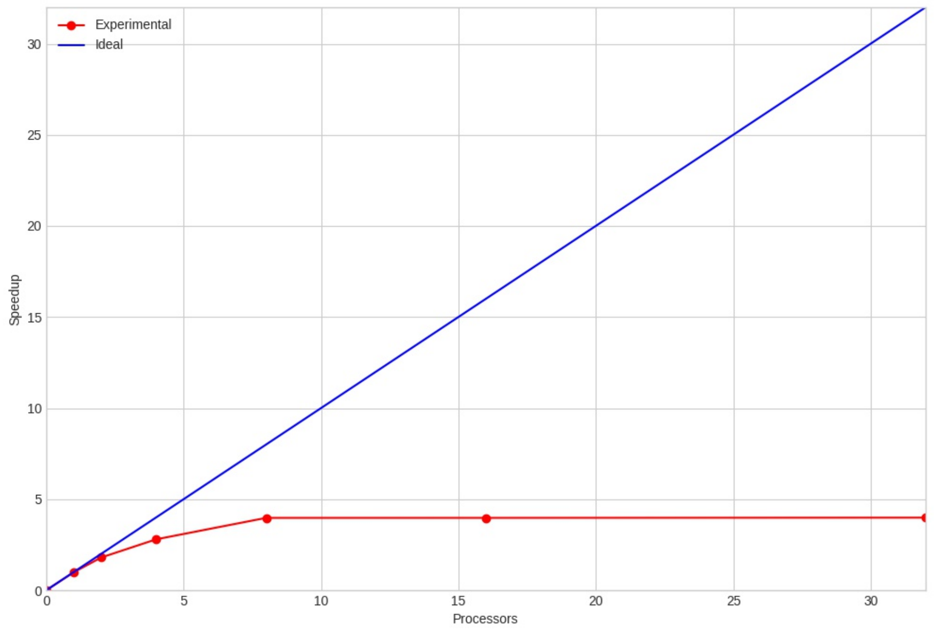
2.2.6 Size-50000-O3

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,000 26	6,784 77	6,785 13	0,000 00	6,785 50	1,000 00	1,000 00
Parallel	1	0,000 27	5,904 43	5,905 63	0,000 00	5,906 06	1,148 91	1,148 91
Parallel	2	0,000 56	6,389 53	6,391 61	0,000 00	3,197 02	2,122 44	1,061 22
Parallel	4	0,000 81	8,118 24	8,122 13	0,000 00	2,035 69	3,333 26	0,833 32
Parallel	8	0,001 42	11,473 42	11,489 02	0,001 57	1,439 90	4,712 47	0,589 06
Parallel	16	0,006 13	11,393 17	11,396 32	0,001 09	1,451 59	4,674 52	0,292 16
Parallel	32	0,006 67	11,404 05	11,409 37	0,001 17	1,444 67	4,696 93	0,146 78



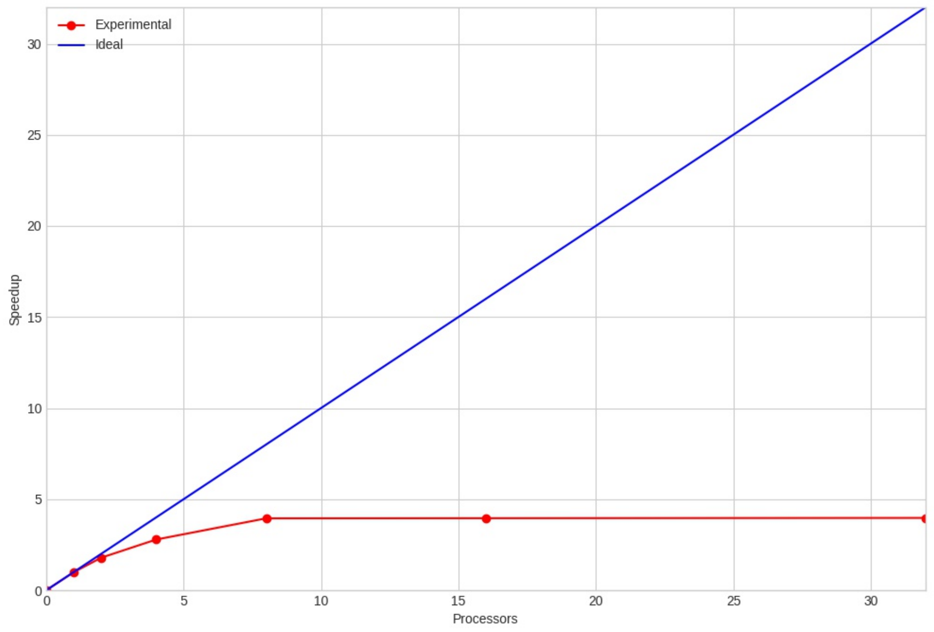
2.2.7 Size-80000-O1

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,000 42	14,680 52	14,679 94	0,000 92	14,682 83	1,000 00	1,000 00
Parallel	1	0,000 42	14,853 39	14,854 44	0,000 00	14,857 71	0,988 23	0,988 23
Parallel	2	0,000 85	16,307 90	16,319 38	0,000 00	8,165 29	1,798 20	0,899 10
Parallel	4	0,001 10	20,930 15	20,934 45	0,000 00	5,252 10	2,795 61	0,698 90
Parallel	8	0,002 20	29,201 90	29,204 05	0,001 40	3,696 21	3,972 41	0,496 55
Parallel	16	0,009 38	29,225 01	29,236 16	0,001 80	3,695 34	3,973 33	0,248 33
Parallel	32	0,009 53	29,269 99	29,276 37	0,003 49	3,684 05	3,985 51	0,124 55



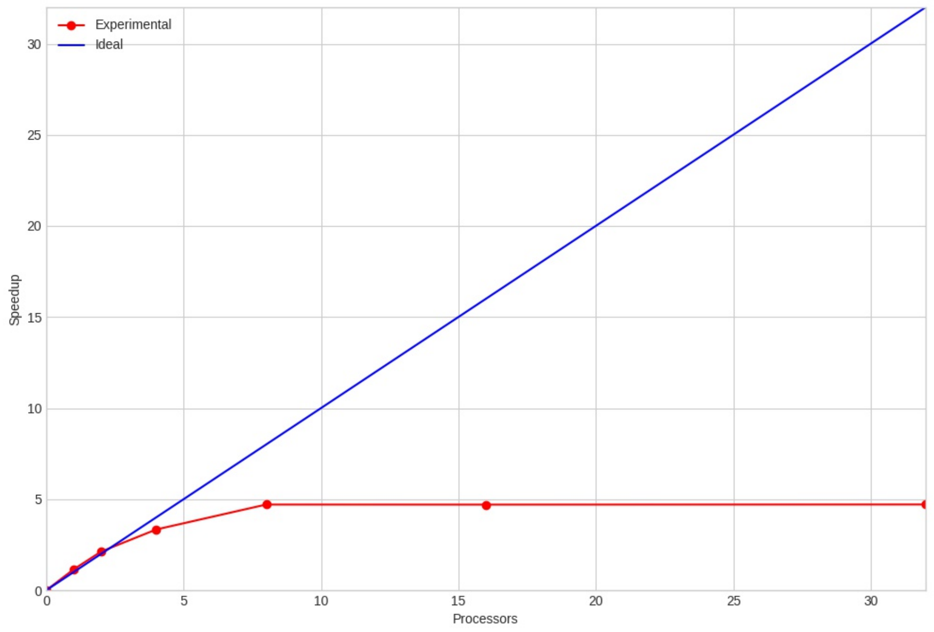
2.2.8 Size-80000-O2

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,000 41	14,377 09	14,377 29	0,000 00	14,378 81	1,000 00	1,000 00
Parallel	1	0,000 42	14,483 79	14,483 93	0,000 00	14,488 19	0,992 45	0,992 45
Parallel	2	0,001 00	15,882 82	15,883 84	0,000 00	8,045 75	1,787 13	0,893 57
Parallel	4	0,001 09	20,288 69	20,290 24	0,000 52	5,165 29	2,783 73	0,695 93
Parallel	8	0,003 33	28,904 70	28,918 25	0,002 30	3,643 69	3,946 22	0,493 28
Parallel	16	0,009 35	28,795 51	28,801 77	0,003 07	3,636 84	3,953 65	0,247 10
Parallel	32	0,009 86	28,768 07	28,773 68	0,003 18	3,621 63	3,970 26	0,124 07



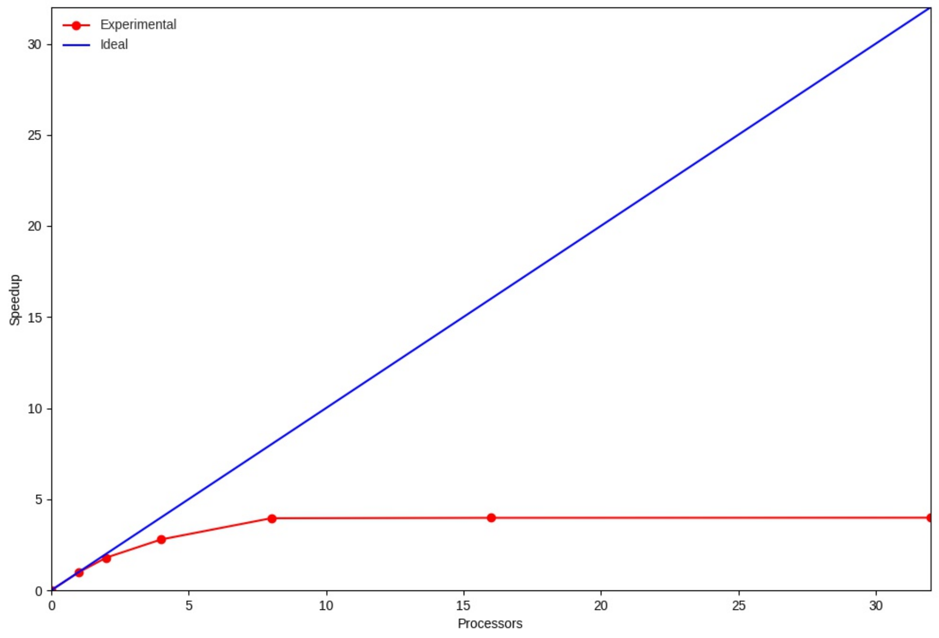
2.2.9 Size-80000-O3

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,000 41	17,384 28	17,384 29	0,000 00	17,388 40	1,000 00	1,000 00
Parallel	1	0,000 42	15,126 28	15,126 85	0,000 00	15,128 53	1,149 38	1,149 38
Parallel	2	0,000 70	16,353 96	16,356 31	0,000 00	8,180 79	2,125 51	1,062 76
Parallel	4	0,001 21	20,820 72	20,825 84	0,000 00	5,214 43	3,334 67	0,833 67
Parallel	8	0,002 66	29,502 68	29,525 16	0,001 21	3,696 35	4,704 21	0,588 03
Parallel	16	0,009 34	29,335 93	29,342 22	0,001 62	3,698 84	4,701 04	0,293 82
Parallel	32	0,009 52	29,332 82	29,338 78	0,002 09	3,693 30	4,708 09	0,147 13



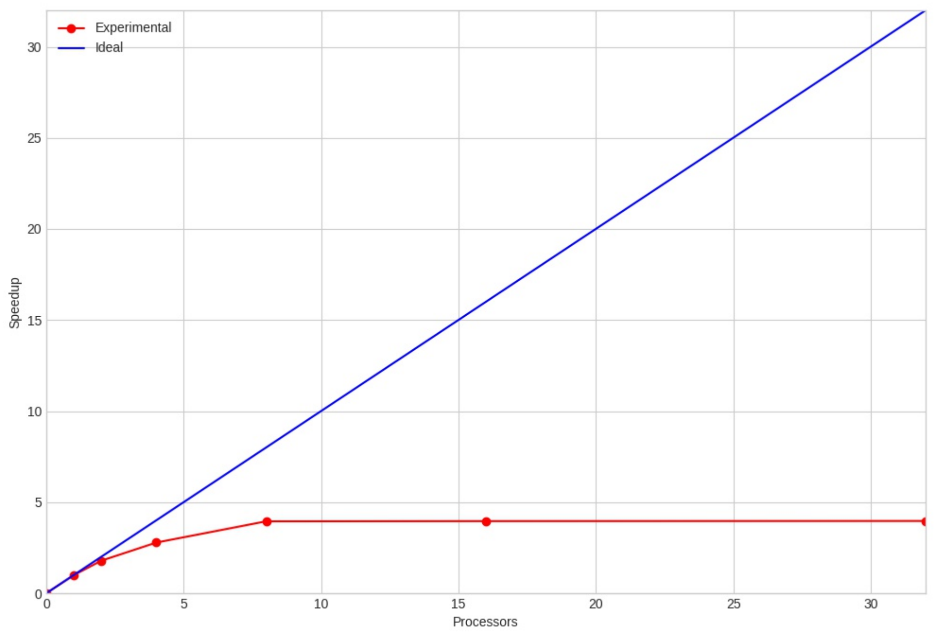
2.2.10 Size-100000-O1

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,000 51	22,921 18	22,922 66	0,000 00	22,923 03	1,000 00	1,000 00
Parallel	1	0,000 52	23,217 05	23,217 72	0,000 00	23,220 00	0,987 21	0,987 21
Parallel	2	0,001 07	25,599 70	25,602 39	0,000 00	12,810 16	1,789 44	0,894 72
Parallel	4	0,001 51	32,789 13	32,793 56	0,000 00	8,222 00	2,788 01	0,697 00
Parallel	8	0,002 84	45,687 47	45,693 13	0,002 81	5,800 20	3,952 11	0,494 01
Parallel	16	0,011 10	45,723 32	45,752 05	0,002 55	5,767 44	3,974 56	0,248 41
Parallel	32	0,011 73	45,779 49	45,788 47	0,003 02	5,756 51	3,982 10	0,124 44



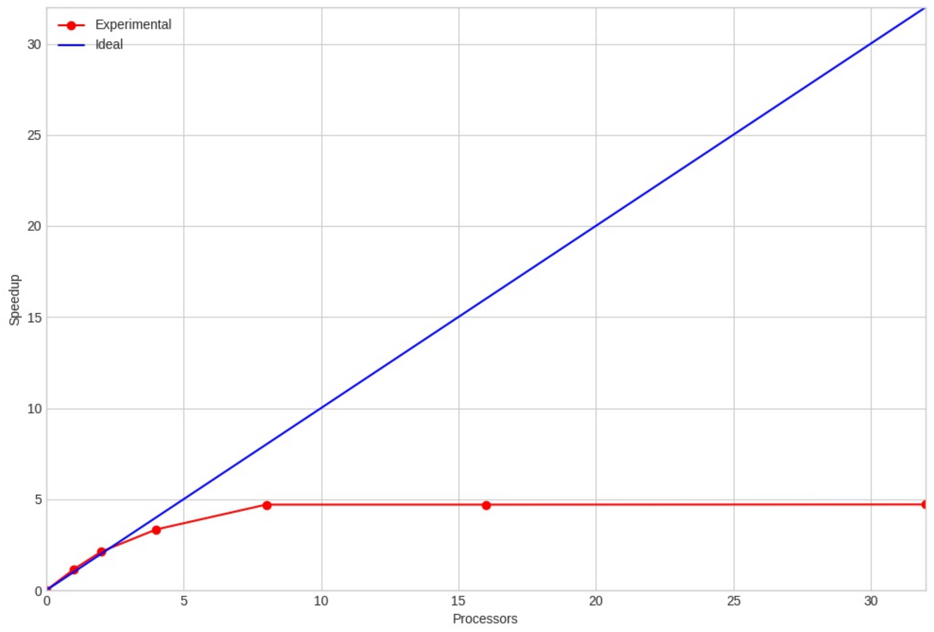
2.2.11 Size-100000-O2

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,000 51	22,466 66	22,467 28	0,000 00	22,468 86	1,000 00	1,000 00
Parallel	1	0,000 52	22,634 51	22,634 76	0,000 00	22,640 50	0,992 42	0,992 42
Parallel	2	0,001 41	24,840 89	24,841 98	0,000 00	12,585 42	1,785 31	0,892 65
Parallel	4	0,001 47	31,751 08	31,753 18	0,000 83	8,081 02	2,780 45	0,695 11
Parallel	8	0,001 88	45,105 93	45,111 98	0,001 49	5,692 08	3,947 39	0,493 42
Parallel	16	0,011 50	45,035 91	45,043 28	0,002 93	5,676 87	3,957 96	0,247 37
Parallel	32	0,012 00	45,090 60	45,096 83	0,004 00	5,664 44	3,966 65	0,123 96



2.2.12 Size-100000-O3

Version	Threads	Init	Counting Sort	User	Sys	Elapsed	Speedup	Efficiency
Serial	1	0,000 51	27,174 22	27,174 86	0,000 00	27,176 00	1,000 00	1,000 00
Parallel	1	0,000 52	23,634 69	23,635 62	0,000 00	23,636 00	1,149 77	1,149 77
Parallel	2	0,001 10	25,629 49	25,632 50	0,000 00	12,820 55	2,119 72	1,059 86
Parallel	4	0,001 51	32,510 44	32,514 55	0,000 00	8,144 36	3,336 79	0,834 20
Parallel	8	0,003 73	46,128 25	46,144 27	0,001 62	5,781 47	4,700 53	0,587 57
Parallel	16	0,011 42	45,956 00	45,964 23	0,003 09	5,781 51	4,700 50	0,293 78
Parallel	32	0,011 77	45,940 96	45,950 78	0,002 15	5,770 50	4,709 47	0,147 17



Chapter 3

Considerations

3.1 First case of study

The first case concerned the performance analysis of the algorithm with time complexity $O(n)$. The chart highlights a different trend compared to the expected one. The trend at first follows the ideal line, but instead of increasing, it falls down and then goes almost constant. This could be expleined by looking at the source code. Beside the initializations of the arrays and the final copy from one array to another, the operations done in this algorithm couldn't have been parallelized.
This creates a bottleneck to the parallel execution of the program as the trend of speedup is affected by this sequential part.

3.2 Second case of study

The second case of study has been compiled and run under the same conditions of the first case, considering the same data type to have a better chance to compare the two cases.
In this case it is clear that, despite the time complexity is $O(n^2)$, so the time spent to execute the code is longer than the first case, parallelization seems to be more effective in this solution. Also in this case it could be seen that parallelization with 8 threads is the best case found in the analysis.
The charts highlight the increasing trend of speedup from 1 to 8 threads , but it also could be seen that it remains almost constant after 8 threads. In conclusion we can assume that the best compromise on this hardware configuration is with 8 threads.

3.3 Other considerations

During the project, the scheduling of threads created has been left to the OMP library, instead of choosing an explicit specification of schedule types. Other optimization could have been possible such as loop unrolling with nested for loops in order to boost program performances, but by optimizing the program with gcc directives (-O2, -O3) such optimization is already adopted. Finally, another type of optimization used is to save the defined variable ELEMENT_TYPE in cache in order to make its access faster to the program.

3.4 Final considerations

Another function which has not been inspected yet is generateArray, which initialises the array using the pseudorandomic numbre generator rand_r. It has been chosen to use this function instead of rand() as it provides a thread-safe solution, so that parallelization will not affect number generation. It could happen, instead, that, depending the seed on time() function, numbers generated in the same instant cuold have the same value.
Secondly it was also chosen to use huge payloads as instead the extimated times were too low, almost 10^{-6}, so it was necessary to increaser the number of data processed in order to have useful time measures.

Chapter 4

API

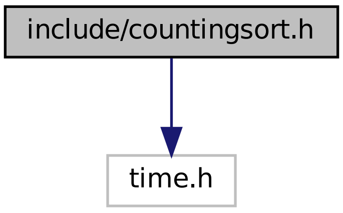
The documentation has been generated through Doxygen. In the following pages we provide the documentation for the public functions defined in the source codes.

File Documentation

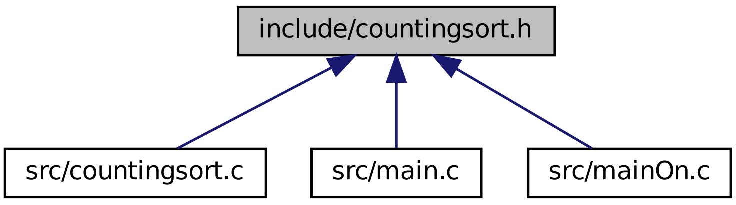
2.1 include/countingsort.h File Reference

```
#include <time.h>
```

Include dependency graph for countingsort.h:



This graph shows which files directly or indirectly include this file:



Macros

- #define STARTTIME(id)
- #define ENDTIME(id, x)

Functions

- void generateArray (ELEMENT_TYPE ∗, int, int)

  The function is used to initiaslize with pseudorandomic numbers the array. This is a thread safe version as rand_r function is used instead of rand().
- void countSortOn2 (ELEMENT_TYPE ∗, int, int)

  The function is used to sort the array my_array according to the algorithm counting sort. This function provides an implementation with a time complexity of O(n^2), being so less convenient than another version of this algorithm in the sequencial performances.
- void countSortOn (ELEMENT_TYPE ∗, ELEMENT_TYPE ∗, int, int)

  This function sorts an array according to the counting sort algorithm using optimized loops with optimized for loop. This function has a time complexity O(n) being in the sequential version more convenien than the previous version.

2.1.1 Macro Definition Documentation

2.1.1.1 ENDTIME

```
#define ENDTIME(
              id,
              x )
```

Value:

```
end_time_42_##id = clock(); \
x = ((double)(end_time_42_##id - start_time_42_##id)) / CLOCKS_PER_SEC
```

2.1.1.2 STARTTIME

```
#define STARTTIME(
              id )
```

Value:

```
clock_t start_time_42_##id, end_time_42_##id; \
start_time_42_##id = clock()
```

macros to get execution time: both macros have to be in the same scope define a double variable to use in ENDTIME before STARTTIME: double x; the variable will hold the execution time in seconds.

2.1.2 Function Documentation

2.1.2.1 countSortOn()

```
void countSortOn (
            ELEMENT_TYPE * my_array,
            ELEMENT_TYPE * temp,
            int length,
            int threads )
```

This function sorts an array according to the counting sort algorithm using optimized loops with optimized for loop. This function has a time complexity O(n) being in the sequential version more convenien than the previous version.

Parameters

my_array	a pointer to an array which must be sorted.
temp	the pointer to the array which must be sorted.
length	size of my_array.
threads	number of threads.

2.1.2.2 countSortOn2()

```
void countSortOn2 (
            ELEMENT_TYPE * my_array,
            int length,
            int threads )
```

The function is used to sort the array my_array according to the algorithm counting sort. This function provides an implementation with a time complexity of O(n^2), being so less convenient than another version of this algorithm in the sequencial performances.

Parameters

my_array	the pointer to the array to be sorted.
length	size of my_array.
threads	number of threads.

2.1.2.3 generateArray()

```
void generateArray (
            ELEMENT_TYPE * my_array,
            int length,
            int threads )
```

The function is used to initiaslize with pseudorandomic numbers the array. This is a thread safe version as rand_r function is used instead of rand().

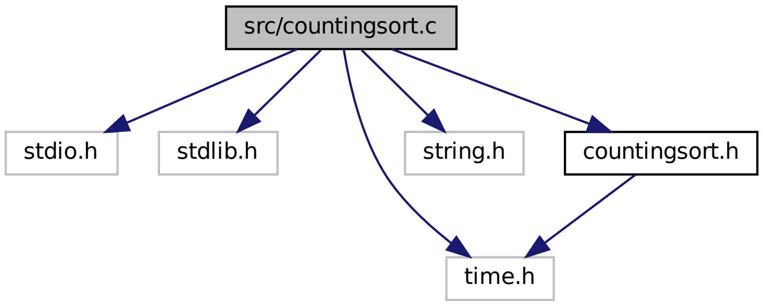
Parameters

my_array	the pointer to the memory area of the array
length	size of my_array.
threads	the number of threads.

2.2 src/countingsort.c File Reference

```
#include <stdio.h>
```

```
#include <stdlib.h>
#include <time.h>
#include <string.h>
#include "countingsort.h"
```
Include dependency graph for countingsort.c:



Macros

- #define omp_get_thread_num() 0

Functions

- void generateArray (ELEMENT_TYPE ∗my_array, int length, int threads)

  The function is used to initiaslize with pseudorandomic numbers the array. This is a thread safe version as rand_r function is used instead of rand().
- void countSortOn2 (ELEMENT_TYPE ∗my_array, int length, int threads)

  The function is used to sort the array my_array according to the algorithm counting sort. This function provides an implementation with a time complexity of O(n^2), being so less convenient than another version of this algorithm in the sequencial performances.
- void countSortOn (ELEMENT_TYPE ∗my_array, ELEMENT_TYPE ∗temp, int length, int threads)

  This function sorts an array according to the counting sort algorithm using optimized loops with optimized for loop. This function has a time complexity O(n) being in the sequential version more convenien than the previous version.

2.2.1 Macro Definition Documentation

2.2.1.1 omp_get_thread_num

```
#define omp_get_thread_num( ) 0
```

2.2.2 Function Documentation

2.2.2.1 countSortOn()

```
void countSortOn (
            ELEMENT_TYPE * my_array,
            ELEMENT_TYPE * temp,
            int length,
            int threads )
```

This function sorts an array according to the counting sort algorithm using optimized loops with optimized for loop. This function has a time complexity O(n) being in the sequential version more convenien than the previous version.

Parameters

my_array	a pointer to an array which must be sorted.
temp	the pointer to the array which must be sorted.
length	size of my_array.
threads	number of threads.

2.2.2.2 countSortOn2()

```
void countSortOn2 (
            ELEMENT_TYPE * my_array,
            int length,
            int threads )
```

The function is used to sort the array my_array according to the algorithm counting sort. This function provides an implementation with a time complexity of O(n^2), being so less convenient than another version of this algorithm in the sequencial performances.

Parameters

my_array	the pointer to the array to be sorted.
length	size of my_array.
threads	number of threads.

2.2.2.3 generateArray()

```
void generateArray (
            ELEMENT_TYPE * my_array,
            int length,
            int threads )
```

The function is used to initiaslize with pseudorandomic numbers the array. This is a thread safe version as rand_r function is used instead of rand().

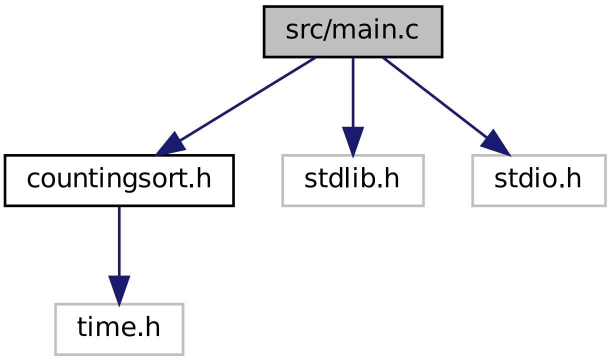
Parameters

my_array	the pointer to the memory area of the array
length	size of my_array.
threads	the number of threads.

2.3 src/main.c File Reference

```
#include "countingsort.h"
#include <stdlib.h>
#include <stdio.h>
```

Include dependency graph for main.c:



Functions

- int main (int argc, char const ∗argv[ ])

2.3.1 Function Documentation

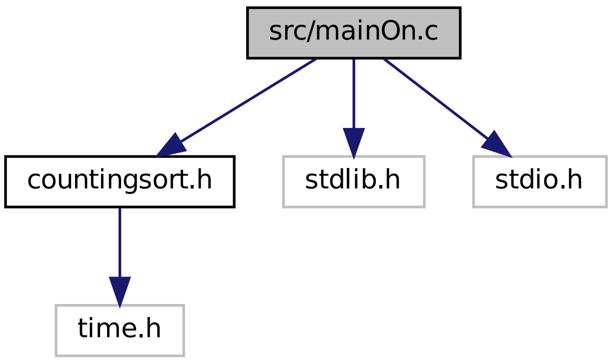
2.3.1.1 main()

```
int main (
            int argc,
            char const * argv[ ] )
```

2.4 src/mainOn.c File Reference

```
#include "countingsort.h"
#include <stdlib.h>
#include <stdio.h>
```

Include dependency graph for mainOn.c:



Functions

- int main (int argc, char const ∗argv[ ])

2.4.1 Function Documentation

2.4.1.1 main()

```
int main (
            int argc,
            char const * argv[] )
```

Chapter 5

How to run

1. Before running the execution make sure that you have the permission as administrator to run the example. To prevent possible errors it has been insert a command that gives right permissions to run the code properly.

```
chmod 777 -R *
mkdir build
cd build
cmake ..
```

2. Generate executables with

```
make
```

3. To generate measures:

    - To generate measures for the execution with time complexity $O(n^2)$ run

    ```
    make generate_measuresOn2
    ```

    - To generate measures run for the execution with time complexity $O(n)$

    ```
    make generate_measuresOn
    ```

4. To extract measures:

    - To extract mean times and speedup curves for the execution with time complexity $O(n^2)$ them run

    ```
    make extract_measuresOn2
    ```

- To extract mean times and speedup curves for the execution with time complexity $O(n)$ them run

  ```
  make extract_measuresOn
  ```

Results can be found in the measures/measureOn (first case) or in measures/measure (second case) directory, divided by problem size and the gcc optimization option used.

In the current project you generate measures for int type, if you want to generate measures for double type, it could be dove just for the second case and it coould be done by changing the word "double" in "int" in the file CMakeLists.txt.

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

The counting sort function was taken here:

For the first case of study:

https://en.wikipedia.org/wiki/Counting_sort

For the second case of study:

https://github.com/ianliu/programacao-paralela/blob/master/omp-count-sort/main.c