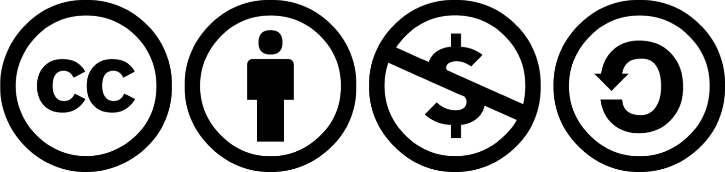
Università degli Studi di Salerno  
DIEM, a.a 2021-22  
Course: HIGH PERFORMANCE COMPUTING  
Lecturer: Francesco Moscato



*Salvatore Grimaldi* 0622701742 s.grimaldi29@studenti.unisa.it  
*Enrico Maria Di Mauro* 0622701706 e.dimauro5@studenti.unisa.it  
*Allegra Cuzzocrea* 0622701707 a.cuzzocrea2@studenti.unisa.it

PARALLELIZATION AND PERFORMANCES EVALUATION  
OF ***COUNTING SORT***  
ALGORITHM WITH **MPI**

Index

[Introduction 3](#_Toc95148796)

[Problem description 4](#_Toc95148797)

[Theoretical notes 5](#_Toc95148798)

[Experimental setup 6](#_Toc95148799)

[Hardware 6](#_Toc95148800)

[CPU 6](#_Toc95148801)

[RAM 11](#_Toc95148802)

[Software 11](#_Toc95148803)

[Case studies and performances 12](#_Toc95148804)

[Graphic and table legend 12](#_Toc95148805)

[Case Study n.1 13](#_Toc95148806)

[SIZE-1mln-RANGE-1k-OPT-O0 14](#_Toc95148807)

[SIZE-1mln-RANGE-1k-OPT-O1 15](#_Toc95148808)

[SIZE-1mln-RANGE-1k-OPT-O2 16](#_Toc95148809)

[SIZE-1mln-RANGE-1k-OPT-O3 17](#_Toc95148810)

[SIZE-1mln-RANGE-100k-OPT-O0 18](#_Toc95148811)

[SIZE-1mln-RANGE-100k-OPT-O1 19](#_Toc95148812)

[SIZE-1mln-RANGE-100k-OPT-O2 20](#_Toc95148813)

[SIZE-1mln-RANGE-100k-OPT-O3 21](#_Toc95148814)

[SIZE-10mln-RANGE-1k-OPT-O0 22](#_Toc95148815)

[SIZE-10mln-RANGE-1k-OPT-O1 23](#_Toc95148816)

[SIZE-10mln-RANGE-1k-OPT-O2 24](#_Toc95148817)

[SIZE-10mln-RANGE-1k-OPT-O3 25](#_Toc95148818)

[SIZE-10mln-RANGE-100k-OPT-O0 26](#_Toc95148819)

[SIZE-10mln-RANGE-100k-OPT-O1 27](#_Toc95148820)

[SIZE-10mln-RANGE-100k-OPT-O2 28](#_Toc95148821)

[SIZE-10mln-RANGE-100k-OPT-O3 29](#_Toc95148822)

[Case Study n.2 30](#_Toc95148823)

[SIZE-1mln-RANGE-1k-OPT-O0 31](#_Toc95148824)

[SIZE-1mln-RANGE-1k-OPT-O1 32](#_Toc95148825)

[SIZE-1mln-RANGE-1k-OPT-O2 33](#_Toc95148826)

[SIZE-1mln-RANGE-1k-OPT-O3 34](#_Toc95148827)

[SIZE-1mln-RANGE-100k-OPT-O0 35](#_Toc95148828)

[SIZE-1mln-RANGE-100k-OPT-O1 36](#_Toc95148829)

[SIZE-1mln-RANGE-100k-OPT-O2 37](#_Toc95148830)

[SIZE-1mln-RANGE-100k-OPT-O3 38](#_Toc95148831)

[SIZE-10mln-RANGE-1k-OPT-O0 39](#_Toc95148832)

[SIZE-10mln-RANGE-1k-OPT-O1 40](#_Toc95148833)

[SIZE-10mln-RANGE-1k-OPT-O2 41](#_Toc95148834)

[SIZE-10mln-RANGE-1k-OPT-O3 42](#_Toc95148835)

[SIZE-10mln-RANGE-100k-OPT-O0 43](#_Toc95148836)

[SIZE-10mln-RANGE-100k-OPT-O1 44](#_Toc95148837)

[SIZE-10mln-RANGE-100k-OPT-O2 45](#_Toc95148838)

[SIZE-10mln-RANGE-100k-OPT-O3 46](#_Toc95148839)

[Considerations 47](#_Toc95148840)

[Case study n.1 47](#_Toc95148841)

[Considerations about counting sort 47](#_Toc95148842)

[Considerations about initialization 47](#_Toc95148843)

[Case study n.2 48](#_Toc95148844)

[Considerations about counting sort 48](#_Toc95148845)

[Considerations about initialization 48](#_Toc95148846)

[Further considerations 48](#_Toc95148847)

[API – Case study n.1 49](#_Toc95148848)

[Public functions mainS.c 49](#_Toc95148849)

[Public functions countingsort.c 49](#_Toc95148850)

[Public functions mainS.c documentation 49](#_Toc95148851)

[function init 49](#_Toc95148852)

[function countingSort 49](#_Toc95148853)

[Public functions countingsort.c documentation 50](#_Toc95148854)

[function init 50](#_Toc95148855)

[function countingSort 50](#_Toc95148856)

[API – Case study n.2 51](#_Toc95148857)

[Public functions mainS.c 51](#_Toc95148858)

[Public functions countingsort.c 51](#_Toc95148859)

[Public functions mainS.c documentation 51](#_Toc95148860)

[function init 51](#_Toc95148861)

[function countingSort 52](#_Toc95148862)

[function readingFile 52](#_Toc95148863)

[Public functions countingsort.c documentation 52](#_Toc95148864)

[function init 52](#_Toc95148865)

[function countingSort 53](#_Toc95148866)

[function readingFile 53](#_Toc95148867)

[How to run 54](#_Toc95148868)

# Introduction

The main purpose of this report is parallelizing and evaluating performances of Counting Sort Algorithm. MPI (Message Passing Interface) is the communication protocol for parallel programming used in this report.

In the following pages the problem faced is going to be described in a detailed way, paying great attention to the description of the case studies considered. The results are going to be analysed and explained in the light of the theoretical knowledge acquired during the High-Performance Computing course held by prof. Francesco Moscato at University of Salerno.

# Problem description

The problem is how to parallelize and evaluate performances of **Counting Sort Algorithm**, by using **MPI**. Counting sort is an integer sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array that is then used to get the actual sorted array.

Counting sort can be used only to sort collections of objects whose keys are positive integers: this is the reason why it is described as an integer sorting algorithm. Counting sort is not a comparison sort, which means that it is not based on comparisons among the objects of the collection meant to be sorted. Counting sort running time is linear in the number of items and the difference between the maximum key value and the minimum key value, so it is only suitable for direct use in situations where the variation in keys is not significantly greater than the number of items. Moreover, to make things easier and faster to understand, it has been chosen to deal with a collection that is a simple array of integers.

Counting sort algorithm can be divided in few steps:

1. A for-loop determines the minimum (**min**) and the maximum (**max**) integers inside the unsorted array **a**.
2. An auxiliary array **c**, whose length is is allocated.
3. A for-loop initializes to 0 all **c** elements.
4. A for loop traverses **a** and at each iteration increments by 1 the **c** element placed in the position corresponding to the visited **a** element.
5. A for loop increments every **c** element by adding to it the sum of all its previous elements in **c**.
6. Two nested for-loops exploit **c** content in order to sort in place array **a**.

Parallelization is obtained thanks to MPI. It is a standardized and portable message-passing standard designed to function on parallel computing architectures. The MPI standard defines the syntax and semantics of library routines that are useful to a wide range of users writing portable message-passing programs in C, C++, and Fortran (in this report we deal with C language).

There are several open-source MPI implementations, such as **MPICH** (the one used for this project), OpenMPI and Intel MPI. Message Passing Interface is based on a **Distributed Memory Model**, in which several nodes send and receive messages, so that a sort of memory abstraction is created.

MPI was born to allow parallel computing to improve performances. MPI is not a programming language, but a framework in which a fundamental role is played by the so-called middleware MPI, which manages synchronization among parallel processes and messages mechanism. MPI is mostly thought for physical **clusters** (e.g., raspberry clusters), but ensures good performances even when the program is run by a single machine, if the number of generated processes is not greater than the number of available cores.

One of the main MPI concepts is the meaning of **communicator**: a sort of container of processes, in which each of them is identified by a rank. Communicator size is given by the number of processes it manages and cannot be changed after Communicator creation.

# Theoretical notes

Given that this report purpose is evaluating performances through speed-up and efficiency, it is fundamental to clarify these two words meanings.

* **Speed-up**: it is a number that measures the relative performance of two systems processing the same problem. More technically, it is the improvement in speed of execution of a task executed on two similar architectures with different resources. The notion of speed-up was established by Amdahl's law, which was particularly focused on parallel processing. For example, let's consider she same program executed on N processors and on 1 processor: if the program executed on only 1 processor is K times slower than the same program executed on N processors, we can say that the obtained speed-up is equal to K. In the most of cases, speed-up functions, plotted considering the same algorithm launched on a different number of processors, are not linear but present a particular trend that can be explained in the light of machine characteristics, algorithm complexity, parallelizable portions, framework used, etc.
* **Efficiency**: it is given by the execution time of the algorithm on 1 processor divided by P \* the execution time of the same algorithm on P processors.

Efficiency is a useful measure of parallel algorithm quality.

# Experimental setup

Information provided below refers to hardware and software used during the evaluation of performances.

## Hardware

### CPU

processor : 0

vendor\_id : GenuineIntel

cpu family : 6

model : 142

model name : Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz

stepping : 12

microcode : 0xffffffff

cpu MHz : 1992.006

cache size : 8192 KB

physical id : 0

siblings : 8

core id : 0

cpu cores : 4

apicid : 0

initial apicid : 0

fpu : yes

fpu\_exception : yes

cpuid level : 22

wp : yes

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant\_tsc rep\_good nopl xtopology cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4\_1 sse4\_2 movbe popcnt aes xsave avx f16c rdrand hypervisor lahf\_lm abm 3dnowprefetch invpcid\_single ssbd ibrs ibpb stibp ibrs\_enhanced tpr\_shadow vnmi ept vpid ept\_ad fsgsbase bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves flush\_l1d arch\_capabilities

vmx flags : vnmi invvpid ept\_x\_only ept\_ad ept\_1gb tsc\_offset vtpr ept vpid unrestricted\_guest ept\_mode\_based\_exec

bugs : spectre\_v1 spectre\_v2 spec\_store\_bypass swapgs itlb\_multihit srbds

bogomips : 3984.01

clflush size : 64

cache\_alignment : 64

address sizes : 39 bits physical, 48 bits virtual

power management:

processor : 1

vendor\_id : GenuineIntel

cpu family : 6

model : 142

model name : Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz

stepping : 12

microcode : 0xffffffff

cpu MHz : 1992.006

cache size : 8192 KB

physical id : 0

siblings : 8

core id : 0

cpu cores : 4

apicid : 1

initial apicid : 1

fpu : yes

fpu\_exception : yes

cpuid level : 22

wp : yes

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant\_tsc rep\_good nopl xtopology cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4\_1 sse4\_2 movbe popcnt aes xsave avx f16c rdrand hypervisor lahf\_lm abm 3dnowprefetch invpcid\_single ssbd ibrs ibpb stibp ibrs\_enhanced tpr\_shadow vnmi ept vpid ept\_ad fsgsbase bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves flush\_l1d arch\_capabilities

vmx flags : vnmi invvpid ept\_x\_only ept\_ad ept\_1gb tsc\_offset vtpr ept vpid unrestricted\_guest ept\_mode\_based\_exec

bugs : spectre\_v1 spectre\_v2 spec\_store\_bypass swapgs itlb\_multihit srbds

bogomips : 3984.01

clflush size : 64

cache\_alignment : 64

address sizes : 39 bits physical, 48 bits virtual

power management:

processor : 2

vendor\_id : GenuineIntel

cpu family : 6

model : 142

model name : Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz

stepping : 12

microcode : 0xffffffff

cpu MHz : 1992.006

cache size : 8192 KB

physical id : 0

siblings : 8

core id : 1

cpu cores : 4

apicid : 2

initial apicid : 2

fpu : yes

fpu\_exception : yes

cpuid level : 22

wp : yes

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant\_tsc rep\_good nopl xtopology cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4\_1 sse4\_2 movbe popcnt aes xsave avx f16c rdrand hypervisor lahf\_lm abm 3dnowprefetch invpcid\_single ssbd ibrs ibpb stibp ibrs\_enhanced tpr\_shadow vnmi ept vpid ept\_ad fsgsbase bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves flush\_l1d arch\_capabilities

vmx flags : vnmi invvpid ept\_x\_only ept\_ad ept\_1gb tsc\_offset vtpr ept vpid unrestricted\_guest ept\_mode\_based\_exec

bugs : spectre\_v1 spectre\_v2 spec\_store\_bypass swapgs itlb\_multihit srbds

bogomips : 3984.01

clflush size : 64

cache\_alignment : 64

address sizes : 39 bits physical, 48 bits virtual

power management:

processor : 3

vendor\_id : GenuineIntel

cpu family : 6

model : 142

model name : Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz

stepping : 12

microcode : 0xffffffff

cpu MHz : 1992.006

cache size : 8192 KB

physical id : 0

siblings : 8

core id : 1

cpu cores : 4

apicid : 3

initial apicid : 3

fpu : yes

fpu\_exception : yes

cpuid level : 22

wp : yes

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant\_tsc rep\_good nopl xtopology cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4\_1 sse4\_2 movbe popcnt aes xsave avx f16c rdrand hypervisor lahf\_lm abm 3dnowprefetch invpcid\_single ssbd ibrs ibpb stibp ibrs\_enhanced tpr\_shadow vnmi ept vpid ept\_ad fsgsbase bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves flush\_l1d arch\_capabilities

vmx flags : vnmi invvpid ept\_x\_only ept\_ad ept\_1gb tsc\_offset vtpr ept vpid unrestricted\_guest ept\_mode\_based\_exec

bugs : spectre\_v1 spectre\_v2 spec\_store\_bypass swapgs itlb\_multihit srbds

bogomips : 3984.01

clflush size : 64

cache\_alignment : 64

address sizes : 39 bits physical, 48 bits virtual

power management:

processor : 4

vendor\_id : GenuineIntel

cpu family : 6

model : 142

model name : Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz

stepping : 12

microcode : 0xffffffff

cpu MHz : 1992.006

cache size : 8192 KB

physical id : 0

siblings : 8

core id : 2

cpu cores : 4

apicid : 4

initial apicid : 4

fpu : yes

fpu\_exception : yes

cpuid level : 22

wp : yes

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant\_tsc rep\_good nopl xtopology cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4\_1 sse4\_2 movbe popcnt aes xsave avx f16c rdrand hypervisor lahf\_lm abm 3dnowprefetch invpcid\_single ssbd ibrs ibpb stibp ibrs\_enhanced tpr\_shadow vnmi ept vpid ept\_ad fsgsbase bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves flush\_l1d arch\_capabilities

vmx flags : vnmi invvpid ept\_x\_only ept\_ad ept\_1gb tsc\_offset vtpr ept vpid unrestricted\_guest ept\_mode\_based\_exec

bugs : spectre\_v1 spectre\_v2 spec\_store\_bypass swapgs itlb\_multihit srbds

bogomips : 3984.01

clflush size : 64

cache\_alignment : 64

address sizes : 39 bits physical, 48 bits virtual

power management:

processor : 5

vendor\_id : GenuineIntel

cpu family : 6

model : 142

model name : Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz

stepping : 12

microcode : 0xffffffff

cpu MHz : 1992.006

cache size : 8192 KB

physical id : 0

siblings : 8

core id : 2

cpu cores : 4

apicid : 5

initial apicid : 5

fpu : yes

fpu\_exception : yes

cpuid level : 22

wp : yes

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant\_tsc rep\_good nopl xtopology cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4\_1 sse4\_2 movbe popcnt aes xsave avx f16c rdrand hypervisor lahf\_lm abm 3dnowprefetch invpcid\_single ssbd ibrs ibpb stibp ibrs\_enhanced tpr\_shadow vnmi ept vpid ept\_ad fsgsbase bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves flush\_l1d arch\_capabilities

vmx flags : vnmi invvpid ept\_x\_only ept\_ad ept\_1gb tsc\_offset vtpr ept vpid unrestricted\_guest ept\_mode\_based\_exec

bugs : spectre\_v1 spectre\_v2 spec\_store\_bypass swapgs itlb\_multihit srbds

bogomips : 3984.01

clflush size : 64

cache\_alignment : 64

address sizes : 39 bits physical, 48 bits virtual

power management:

processor : 6

vendor\_id : GenuineIntel

cpu family : 6

model : 142

model name : Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz

stepping : 12

microcode : 0xffffffff

cpu MHz : 1992.006

cache size : 8192 KB

physical id : 0

siblings : 8

core id : 3

cpu cores : 4

apicid : 6

initial apicid : 6

fpu : yes

fpu\_exception : yes

cpuid level : 22

wp : yes

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant\_tsc rep\_good nopl xtopology cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4\_1 sse4\_2 movbe popcnt aes xsave avx f16c rdrand hypervisor lahf\_lm abm 3dnowprefetch invpcid\_single ssbd ibrs ibpb stibp ibrs\_enhanced tpr\_shadow vnmi ept vpid ept\_ad fsgsbase bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves flush\_l1d arch\_capabilities

vmx flags : vnmi invvpid ept\_x\_only ept\_ad ept\_1gb tsc\_offset vtpr ept vpid unrestricted\_guest ept\_mode\_based\_exec

bugs : spectre\_v1 spectre\_v2 spec\_store\_bypass swapgs itlb\_multihit srbds

bogomips : 3984.01

clflush size : 64

cache\_alignment : 64

address sizes : 39 bits physical, 48 bits virtual

power management:

processor : 7

vendor\_id : GenuineIntel

cpu family : 6

model : 142

model name : Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz

stepping : 12

microcode : 0xffffffff

cpu MHz : 1992.006

cache size : 8192 KB

physical id : 0

siblings : 8

core id : 3

cpu cores : 4

apicid : 7

initial apicid : 7

fpu : yes

fpu\_exception : yes

cpuid level : 22

wp : yes

flags : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr sse sse2 ss ht syscall nx pdpe1gb rdtscp lm constant\_tsc rep\_good nopl xtopology cpuid pni pclmulqdq vmx ssse3 fma cx16 pcid sse4\_1 sse4\_2 movbe popcnt aes xsave avx f16c rdrand hypervisor lahf\_lm abm 3dnowprefetch invpcid\_single ssbd ibrs ibpb stibp ibrs\_enhanced tpr\_shadow vnmi ept vpid ept\_ad fsgsbase bmi1 avx2 smep bmi2 erms invpcid rdseed adx smap clflushopt xsaveopt xsavec xgetbv1 xsaves flush\_l1d arch\_capabilities

vmx flags : vnmi invvpid ept\_x\_only ept\_ad ept\_1gb tsc\_offset vtpr ept vpid unrestricted\_guest ept\_mode\_based\_exec

bugs : spectre\_v1 spectre\_v2 spec\_store\_bypass swapgs itlb\_multihit srbds

bogomips : 3984.01

clflush size : 64

cache\_alignment : 64

address sizes : 39 bits physical, 48 bits virtual

power management:

### RAM

MemTotal: 3930548 kB

MemFree: 3242452 kB

MemAvailable: 3273480 kB

Buffers: 51864 kB

Cached: 167144 kB

SwapCached: 0 kB

Active: 98372 kB

Inactive: 323472 kB

Active(anon): 220 kB

Inactive(anon): 202908 kB

Active(file): 98152 kB

Inactive(file): 120564 kB

Unevictable: 0 kB

Mlocked: 0 kB

SwapTotal: 1048576 kB

SwapFree: 1048576 kB

Dirty: 16 kB

Writeback: 0 kB

AnonPages: 202820 kB

Mapped: 64436 kB

Shmem: 284 kB

KReclaimable: 25488 kB

Slab: 56136 kB

SReclaimable: 25488 kB

SUnreclaim: 30648 kB

KernelStack: 4128 kB

PageTables: 8060 kB

NFS\_Unstable: 0 kB

Bounce: 0 kB

WritebackTmp: 0 kB

CommitLimit: 3013848 kB

Committed\_AS: 893480 kB

VmallocTotal:34359738367 kB

VmallocUsed: 23996 kB

VmallocChunk: 0 kB

Percpu: 2624 kB

AnonHugePages: 24576 kB

ShmemHugePages: 0 kB

ShmemPmdMapped: 0 kB

FileHugePages: 0 kB

FilePmdMapped: 0 kB

HugePages\_Total: 0

HugePages\_Free: 0

HugePages\_Rsvd: 0

HugePages\_Surp: 0

Hugepagesize: 2048 kB

Hugetlb: 0 kB

DirectMap4k: 19456 kB

DirectMap2M: 3033088 kB

DirectMap1G: 9437184 kB

## Software

Ubuntu : 20.04.3

GCC : 9.3.0  
MPICH : 3.3.2

# Case studies and performances

In this report two different case studies are considered:

* Case Study n.1: it is characterized by the use of the typical MPI collective primitives. Moreover, files are NOT used.
* Case Study n.2: it is characterized by the use of files, which are particularly interesting in MPI.

In both the case studies the sizes considered are 1mln and 10mln, where sizes are the dimensions of the arrays meant to be sorted. Moreover, two ranges are taken in account: 1k, 100k. The range represents the maximum integer that can be found in the unsorted array.

The choice regarding the use of multiple sizes and ranges is determined by the intention of testing the counting sort algorithm in different conditions. One of the purposes is comparing algorithm performances, in terms of speed-up and efficiency, for different sizes and ranges.

It is important to notice that in the following case studies are represented also the speed-ups obtained thanks to the parallelization of unsorted array initialization.

## Graphic and table legend

The graphic and table legend useful to read the results is reported below:

* **Version**: program version considered (serial or parallel). Each case study has a serial and a parallel version
* **Processes**: number of processes that are involved in program execution
* **TimeInit**: elapsed time for unsorted array initialization
* **TimeCount**: elapsed time for counting sort algorithm
* **Speedup\_timeInit**: speed-up corresponding to the unsorted array initialization portion of the program
* **Efficiency\_timeInit**: efficiency corresponding to the unsorted array initialization portion of the program
* **Speedup\_timeCount**: speed-up corresponding to the counting sort portion of the program
* **Efficiency\_timeCount**: efficiency corresponding to the counting sort portion of the program

***A table and two graphs are provided for each measure. In particular, the first graph represents the trend of initialization speed-up, while the second one represents the trend of sorting speed-up.***

It is relevant to observe that in all speed-up graphs the point corresponding to 1 process is obtained comparing the parallel version execution time (with number of processes equal to 1) to the sequential version execution time. All the other points, instead, are obtained comparing the parallel version execution time (with number of processes equal to N, depending on the point) to the parallel version execution time with 1 process.

## Case Study n.1

To better understand the following results, describing the logic behind the program is very useful.

The sequential version is extremely easy: it takes two parameters as input, which represent size (unsorted array length) and range. Then it calls a function, whose name is *init()* and whose role is initializing randomically the unsorted array. In the end, another function, whose name is *countingSort(),* is called: it just sorts the previous created array using counting sort algorithm.

The parallel version is a bit more complex. There are always two parameters with the same meaning seen above, but before dealing with them it is necessary to initialize the MPI environment. The only process that allocates the entire unsorted array **full\_array**, according to the size inserted as input, is the one with rank 0. Then every process, included the one with rank 0, calls a function named *init()*, which is meant to initialize the unsorted array. This purpose is reached thanks to the following logic: every process must allocate and randomically initialize an array called **piece\_init\_array**. Afterwards, all these arrays are put together and sent to the process with rank 0, which keeps them in **full\_array**.

Finished the initialization, it is counting sort turn: every process calls a function named *countingSort()*. The unsorted array **full\_array** previously created must be scattered among all the available processes, so that each of them can work on a portion called **piece\_of\_array**. The i-th process computes the minimum and the maximum of its **piece\_of\_array**. These results are local so a reduction is compulsory to get the global maximum (**max**) and the global minimum (**min**). After that, the i-th process must allocate an auxiliary array named **c\_local**, whose length is equal to .

Three for-loops are executed: the first one initializes to 0 every **c\_local** element; the second one increments by 1 the **c\_local** element placed in the position corresponding to the visited **piece\_of\_array** element; the third one increments every **c\_local** element by adding to it the sum of all its previous elements in **c\_local**. After that, all the different **c\_local** must be summed together through a reduction: the result is represented by another array, whose name is **c**, allocated only by rank 0 process, which can eventually complete counting sort, executing two for-loops that exploit **c** content to sort **full\_array** in place.

It is important to observe that the job of measuring elapsed time is assigned to the process with rank 0. In fact it is the last one to finish *countingSort()*, and only when it terminates, **full\_array** is actually sorted. TimeInit, instead, is basically the same for all the processes.

The following graphs are obtained considering not only different sizes and ranges, but also different gcc optimizations.

### SIZE-1mln-RANGE-1k-OPT-O0



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-1k-OPT-O1



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-1k-OPT-O2



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-1k-OPT-O3



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-100k-OPT-O0



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-100k-OPT-O1



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-100k-OPT-O2



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-100k-OPT-O3



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-1k-OPT-O0



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-1k-OPT-O1



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-1k-OPT-O2



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-1k-OPT-O3

****

Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-100k-OPT-O0



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-100k-OPT-O1



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-100k-OPT-O2



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-100k-OPT-O3



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

## Case Study n.2

To better understand the following results, describing the logic behind the program is very useful.

The sequential version is easy: it takes three parameters as input, which represent size (unsorted array length), range and the name of the file where to work. Then it calls a function, whose name is *init()* and whose role is initializing randomically the unsorted array and to print it on file. In the end, another function, whose name is *countingSort()*, is called: it reads the unsorted array from the file, sorts it using counting sort algorithm and prints it on the same file, overwriting its previous content.

The parallel version is a bit more complex. There are always three parameters with the same meaning seen above, but before dealing with them it is necessary to initialize the MPI environment. Then every process calls a function named *init()*, which is meant to initialize the unsorted array and to print it on file. This purpose is reached thanks to the following logic: every process must allocate and randomically initialize an array called **piece\_init\_array**. After that the file must be opened and the primitive *MPI\_File\_set\_view()* must be called properly, so that each process can write its **piece\_init\_array** on file.

Finished the initialization, it is counting sort turn: every process calls a function named *countingSort()*. The full unsorted array previously written on file must be read in a distributed way by the processes: each of them reads a specific file portion (different from the ones read by the others) and saves its content into an array called **piece\_of\_array**. The i-th process computes the minimum and the maximum of its **piece\_of\_array**. These results are local so a reduction is compulsory to get the global maximum (**max**) and the global minimum (**min**). After that, the i-th process must allocate an auxiliary array named **c\_local**, whose length is equal to .

Three for-loops are executed: the first one initializes to 0 every **c\_local** element; the second one increments by 1 the **c\_local** element placed in the position corresponding to the visited **piece\_of\_array** element; the third one increments every **c\_local** element by adding to it the sum of all its previous elements in **c\_local**. After that, all the different **c\_local** must be summed together through a reduction: the result is represented by another array, whose name is **c**, allocated only by rank 0 process. Afterwards, the process with rank 0 allocates an array **full\_array** intended to contain the sorted array. Counting sort is completed executing two for-loops that exploit **c** content to fill **full\_array**. In the end, **full\_array** is printed on file, overwriting its previous content, by the process with rank 0.

Just like in Case Study n.1, it is important to observe that the job of measuring elapsed time is assigned to the process with rank 0.

The following graphs are obtained considering not only different sizes and ranges, but also different gcc optimizations.

### SIZE-1mln-RANGE-1k-OPT-O0



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-1k-OPT-O1



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-1k-OPT-O2



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-1k-OPT-O3



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-100k-OPT-O0



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-100k-OPT-O1



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-100k-OPT-O2



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-1mln-RANGE-100k-OPT-O3



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-1k-OPT-O0



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-1k-OPT-O1



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-1k-OPT-O2



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-1k-OPT-O3

****

Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-100k-OPT-O0



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-100k-OPT-O1



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-100k-OPT-O2



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

### SIZE-10mln-RANGE-100k-OPT-O3



Chart, line chart

Description automatically generated

Speedup\_timeInit

Chart, line chart

Description automatically generated

Speedup\_timeCount

# Considerations

## Case study n.1

### Considerations about counting sort

* For SIZE-1mln-RANGE-1k, regardless of gcc optimization, performances are not that good. In fact, data dimension is not very high, so MPI middleware overhead (mostly due to message passing) is not offset by parallel computation advantages.
* For SIZE-1mln-RANGE-100k performances keep being not exciting. O2 optimization is the best.
* For SIZE-10mln-RANGE-1k the results are much better thanks to the increasing of array size. Gcc optimizations give a fundamental contribution in terms of speed-up, guaranteeing mostly in O2 and O3 cases, significant improvements. O3 optimization reaches a peak of about 1.42 when the program is executed on 8 processes.
* SIZE-10mln-RANGE-100k is the configuration with which are obtained the best results without optimization (a speed-up of about 1.33 with 2 processes). The reason can be found obviously in the increasing of the amount of data: greater the size and the range are, greater is the advantage introduced by MPI parallelization. MPI overhead, in this case, is fully recovered and does not represent an important problem anymore. All the optimizations make improvements compared to O0 compilation, guaranteeing excellent performances until 8 processes practically in all the cases. The peak is reached with 8 processes and O3 optimization and is about 1.45.
* Fixing the range and increasing the size, TimeCount increases, as expected.
* Fixing the size and changing the range, TimeCount increases, as expected.

### Considerations about initialization

* For SIZE-1mln-RANGE-1k performances are good but not amazing. Different optimizations can't make a great difference.
* For SIZE-1mln-RANGE-100k the results are practically the same, with the only difference that optimization O2 is clearly the best.
* For SIZE-10mln-RANGE-1k performances are much better. The reason is given by the increased dimension of the array that must be initialized. With all the different optimization options, the speed-up is greater than 2 when the program is launched on 4 and 8 processes. In particular, advanced optimizations guarantee better results: the peak is 2.87 and corresponds to the program compiled with O3 optimization and executed on 8 processes.
* For SIZE-10mln-RANGE-100k the results are practically the same, which means that, fixing array size, changing the range does not imply visible performances alterations in terms of speed-up. The peak is 2.89.
* Fixing the range and increasing the size, TimeInit increases, as expected.
* Fixing the size and changing the range, TimeInit does not change visibly, as expected.

## Case study n.2

### Considerations about counting sort

* For SIZE-1mln-RANGE-1k, performances are not very good. In fact, the not very high  
  data dimension does not allow to offset MPI middleware overhead (mostly due to message passing).

Regarding optimizations, better performances are obtained without optimization  
(O0) and they reach a speedup equal to 1.17 when the program is executed on 2 processes.

* For SIZE-1mln-RANGE-100k performances are quite similar to those with range 1k because of the same array size.
* For SIZE-10mln-RANGE-1k performances begin to be better thanks to the increasing of array size. In particular, O0 optimization guarantees a speedup of 1.34 when the program is executed on 2 processes. Besides, O2 and O3 optimizations slightly make performances worse still ensuring a speed-up pick of 1.13 when the number of processes is 4.
* SIZE-10mln-RANGE-100k is the configuration with which are obtained the best speed-up results (1.29 with 4 processes). This can be explained as Case Study n.1: greater the size and range are, greater the advantage introduced by MPI parallelization is.  
  Differently from Case Study n.1, optimizations do not produce better performances than those produced by O0, except for the case obtained with O3 optimization and executed on 8 processes, which reaches a speed-up value of 1.33.

### Considerations about initialization

* For SIZE-1mln-RANGE-1k performances are good but they are not the best obtained. They reach a speed-up value of 1.5. Optimizations do not improve results.
* For SIZE-1mln-RANGE-100k performances are similar to those of previous configuration. The maximum speed-up value is 1.52 obtained with O0. As previously, optimizations do not improve results.
* For SIZE-10mln-RANGE-1k performances get better: O0 produces a speed-up pick of 1.75 with 4 processes. Differently from Case Study n.1, optimizations do not produce better performances, except for the case obtained with O1 when the program is executed on 8 processes, which reaches a speed-up value of 1.88.
* For SIZE-10mln-RANGE-100k results are slightly better than those given by the previous configuration. O0 gives a speed-up pick of 1.76 with 4 processes.  
  Optimizations improve performances only when the number of launched processes is 8. In particular, the maximum value, 1.92, is obtained by O3.

## Further considerations

* Regarding performances, Case Study n.1 is better than Case Study n.2 for the following reason: it guarantees greater speed-ups and shorter elapsed time for both initialization and counting sort. In particular, the maximum speed-up in terms of counting sort reached by Case Study n.1 is 1.45, while 1.34 is the maximum value guaranteed by Case Study n.2. The difference in terms of initialization is even bigger: *speedup\_timeInit* peak for Case Study n.1 is 2.89, while the one reached by Case Study n.2 is 1.92.
* Performances with 16 processes in both Case Studies are very bad because our system has only 4 physical cores and 8 threads.
* The principal reason why Case Study n.1 is faster than Case Study n.2 is because it does not present any overhead due to file management.

# API – Case study n.1

Public Docs also available [here](version1/documentationHTMLDoxygen/html/index.html)

## Public functions mainS.c

|  |  |
| --- | --- |
| Type | Name |
| void | **init**(int n, int range, int \*full\_array)  *This is the function that initializes randomly the array 'full\_array'.* |
| void | **countingSort**(int n, int \*full\_array)  *This is the function that sorts the array 'full\_array' using Counting Sort Algorithm.* |

## Public functions countingsort.c

|  |  |
| --- | --- |
| Type | Name |
| void | **init**(int n, int n\_ranks, int rank, int range, int \*full\_array)  *This is the function that initializes randomly the array 'full\_array' distributing the computation among the processes.* |
| void | **countingSort**(int n, int n\_ranks, int rank, int \*full\_array)  *This is the function that sorts the array 'full\_array' using Counting Sort Algorithm and distributing the computation among the processes.* |

## Public functions mainS.c documentation

### function init

void init(  
 int n,  
 int range,  
 int \*full\_array  
)

**Parameters**:

* n number of array elements.
* range maximum acceptable integer.
* full\_array pointer to the array.

### function countingSort

void countingSort(  
 int n,   
 int \*full\_array  
)

**Parameters**:

* n number of array elements.
* full\_array pointer to the unsorted array.

## Public functions countingsort.c documentation

### function init

void init(  
 int n,   
 int n\_ranks,   
 int rank,   
 int range,   
 int \*full\_array  
)

**Parameters**:

* n number of array elements.
* n\_ranks number of ranks.
* rank rank of the current process.
* range maximum acceptable integer.
* full\_array pointer to the array.

### function countingSort

void countingSort(  
 int n,   
 int n\_ranks,  
 int rank,   
 int \*full\_array  
)

**Parameters**:

* n number of array elements.
* n\_ranks number of ranks.
* rank rank of the current process.
* full\_array pointer to the unsorted array.

# API – Case study n.2

Public Docs also available [here](version2/documentationHTMLDoxygen/html/index.html)

## Public functions mainS.c

|  |  |
| --- | --- |
| Type | Name |
| void | **init**(int n, int range, char \*file\_name)  *This is the function that writes in the file ‘file\_name’ ‘n’ integers.* |
| void | **countingSort**(int n, char \*file\_name)  *This is the function that sorts the integers in ‘file\_name’ using Counting Sort Algorithm.* |
| void | **readingFile**(int n, char \*file\_name)  *This is the function that reads the integers in ‘file\_name’ and prints them on stdout.* |

## Public functions countingsort.c

|  |  |
| --- | --- |
| Type | Name |
| void | **init**(int n, int n\_ranks, int rank, int range, char \*file\_name)  *This is the function that writes in the file 'file\_name' 'n' integers distributing the computation among the processes.* |
| void | **countingSort**(int n, int n\_ranks, int rank, char \*file\_name)  *This is the function that sorts the integers in 'file\_name' using Counting Sort Algorithm and distributing the computation among the processes.* |
| void | **readingFile**(int n, int rank, char \*file\_name)  *This is the function that reads the integers in ‘file\_name’ and prints them on stdout.* |

## Public functions mainS.c documentation

### function init

void init(  
 int n,  
 int range,  
 char \*file\_name  
)

**Parameters**:

* n number of array elements.
* range maximum acceptable integer.
* file\_name file name.

### function countingSort

void countingSort(  
 int n,   
 char \*file\_name  
)

**Parameters**:

* n number of array elements.
* file\_name file name.

### function readingFile

void readingFile(  
 int n  
 char \*file\_name  
)

**Parameters**:

* n number of array elements.
* file\_name file name.

## Public functions countingsort.c documentation

### function init

void init(  
 int n,   
 int n\_ranks,  
 int rank,   
 int range,   
 char \*file\_name  
)

**Parameters**:

* n number of array elements.
* n\_ranks number of ranks.
* rank rank of the current process.
* range maximum acceptable integer.
* file\_name file name.

### function countingSort

void countingSort(  
 int n,   
 int n\_ranks,  
 int rank,   
 char \*file\_name  
)

**Parameters**:

* n number of array elements.
* n\_ranks number of ranks.
* rank rank of the current process.
* file\_name file name.

### function readingFile

void readingFile(  
 int n,  
 int rank,  
 char \*file\_name  
)

**Parameters**:

* n number of array elements.
* rank rank of the current process.
* file\_name file name.

# How to run

Since 2 versions of Counting Sort are provided, you must use command  
cd version1 or cd version2 in order to choose the version to run and:

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\_output  
   ***Attention****: it takes a lot of time. This is the reason why our measures are already included, so you should skip this step.*
4. To extract mean times and speedup curves from them run make extract\_measures

Results can be found in the measure/YYYY-MM-DD.hh:mm:ss directory, divided by problem size and the gcc optimization option used.

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of this license, visit  
<http://creativecommons.org/licenses/by-nc-sa/4.0/> or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.