High Performance Computing

Counting Sort with OpenMP

A.Y. 2021/22

Version 1.0.0

Alessio Pepe	0622701463	a.pepe108@studenti.unisa.it
Teresa Tortorella	0622701507	t.tortorella3@studenti.unisa.it
Paolo Mansi	0622701542	p.mansi5@studenti.unisa.it

1 Index

2 Problem Description

2.1 Assignment

Parallelize and Evaluate Performances of "COUNTING SORT" Algorithm, by using OpenMP.

2.2 Algorithm description

Counting sort is a sorting algorithm that sorts the elements of an array of integer value by counting the number of occurrences of each unique element in the array. The count is stored in an auxiliary array and the sorting is done by mapping the count as an index of the auxiliary array.

The meta-algorithm of the counting sort is the following (from https://it.wikipedia.org/wiki/Counting_sort):

```
countingSort(A[])
   //Calcolo degli elementi max e min
   max ← A[0]
   min ← A[0]
   for i ← 1 to length[A] do
      if (A[i] > max) then
         max ← A[i]
      else if(A[i] < min) then
         min ← A[i]
   end for

   * creates a C array of size max - min + 1
   for i ← 0 to length[C] do
      C[i] ← 0
   end for
   for i ← 0 to length[A] do
      C[A[i] - min] = C[A[i] - min] + 1
   end for
   k ← 0
   for i ← 0 to length[C] do
      while C[i] > 0 do
         A[k] ← i + min
         k ← k + 1
         C[i] ← C[i] - 1
   end for
```

The algorithm consists of three parts:

- find the maximum and minimum value within the array to calculate the width of the range of values.
- use an auxiliary array to counter the frequency of each single value.
- repopulate the array in an orderly manner based on the newly calculated frequency array.

The computational complexity of the algorithm is given by the sum of the computational complexities of the three parts and is in total

$$\big(O(n)\big) + \big(O(n) + O(k)\big) + \big(O(n)\big)$$

where:

- n is the number of elements of the array.
- k is the size of the range of elements in the array (we are considering on average that the allocation of a block of memory costs $O(1)$.

We are aware that it is not possible to parallelize the PMF vector allocation operation and therefore the theoretical speedup curve should not be that of the images that will be shown in the solutions section later, but since this algorithm makes sense to be used when $k \ll n$ then we consider that represented a good approximation.

2.3 Solution idea

2.3.1 Min-max

In implementing this part of the algorithm with OpenMP we manually implemented the reduction mechanism as for some reason it is not supported by our version of the library despite the use of gcc9 (and therefore OpenMP 4.5).

Inside the parallel block we define two local variables for maximum and minimum, after which we use OpenMP to section the array area used by each thread to calculate the maximum and minimum of that area. Finally, we use a critical section so that each thread can update the global maximum and minimum. The computational complexity of this solution is ideally

$$O\left(\frac{n}{p}\right)$$

2.3.2 Counting occurrence

To calculate the vector of PMF we have provided two possible solutions.

In the first solution we allocate a local vector for each thread that will use it to calculate the element frequencies of its part of the array. Next there is a critical section to make sure that each thread can update the global frequency vector. The computational complexity of first solution is ideally

$$O(k) + O\left(\frac{n}{p}\right) + O(k \cdot p)$$

A second solution was to try using an atomic block to make the frequency increment operation thread safe. The computational complexity of second solution is ideally

$$O\left(\frac{n}{p}\right) + O(k)$$

2.3.3 Populating array

To repopulate the array, we have provided two possible solutions.

The first solution involves calculating sequentially, using the main thread, the vector of the CDF starting from that of the PMF to allow the threads to know the position of the array in which the elements must be inserted. In fact, subsequently, OpenMP is used to partition the area that each array must repopulate. The computational complexity of first solution is ideally

$$O(k) + O\left(\frac{n}{p}\right)$$

The second solution is like the first, but we tried to calculate the vector of the CDF in a parallelized way. The computational complexity of second solution is ideally

$$O\left(\frac{k}{p}\right) + O\left(\frac{n}{p}\right)$$

2.3.4 Total solution

We have provided two files counting_sort.h and counting_sort.c which contain the implementations of the sequential and parallel version of the algorithm. In the .c file some macros are set that allow you to choose which of the combinations you want to execute the algorithm to obtain the 4 possible solutions we have analyzed. These have been left set to launch the best solution we have analyzed.

So, there are 4 solutions, which in the file will be named 1-1, 1-2, 2-1 and 2-2 and represent the combinations between the above-mentioned algorithms.

3 How to Run

3.1 Organization of the repository

The files containing the source code, the scripts, the license, and everything associated with it can be viewed at the link https://github.com/pepealessio/countingsort_openmp and have been provided together with the following document.

The following directories are present:

- **/include**: it contains all the header files necessary to solve the requested problem.
- **/script:** contains the scripts needed to evaluate the performance of the provided code.
- **/source:** it contains all the source code and the main file needed to solve the problem.
- **/test:** contains the test files, usable with the ctest framework (or make test), which demonstrate the actual functionality of the provided application.

The following folder is provided for the purpose of demonstrating the authors' measurements, but it can be deleted, or the content will be moved in the folder at the time of application launch:

The following folders are not present in the directory provided, but will be generated when launching the application:

- **/.oldmeasures:** If new measures will be generate when /measures folder already exists, the contents of the measurement folder will be moved to this folder in a subfolder named with the date and time of the move.
- **/.venv:** (will be generated at the script time if the user decide to use a virtual environment as described in the section below). It's just the virtual environment for python3.
- **/build:** Contains all built file with the make command.
- **/docs:** Contains the Doxygen documentation of the source code. The documentation was provided in 2 formats:
    - **/docs/html**
    - **/docs/latex**
- **/measures:** It contains all the raw and processed data of the evaluations (if provided contains measure made on one of the authors' machines). There are the following subfolders:
    - **/measures/plots:** Contains the plots, for each optimization level and size, of the speedup and efficiency as the number of threads used varies, extrapolated from the processed data. Each image has a name structure like *T_ALGO_opt_Y_size_Z.png*, where *Y* indicates the compiler optimization level used for those measurements, and *Z* indicates the size of the problem used for those measurements.
    - **/measures/processed:** Contains data extracted from raw measurements, where speedup and efficiency are compared based on the use of a different number of threads. Each file has a name structure like *T_ALGO_opt_Y_size_Z_range_R.csv*, where *Y* indicates the compiler optimization level used for those measurements, *Z* indicates the size of the problem used for those measurements, and *R* indicates the dimension of the element range of the array.
    - **/measures/raw:** Contains all raw measurements performed by the script. Each file has a name structure like *opt_Y_size_Z_range_R_threads_X.csv*, where *Y* indicates the compiler optimization level used for those measurements, *Z* indicates the size of the problem used for those measurements, and *R* indicates the dimension of the element range of the array and *X* indicate the number of OpenMP's threads used in the application run (0 means sequential application without OpenMP, so with 1 thread).

There are also the following files:

- **.gitignore:** Specifies intentionally untracked files to ignore. We used that for share our work on GitHub.
- **CmakeLists.txt:** contains all the directives needed to compile and test all the necessary executables.
- **Doxyfile:** it contains all the directives to automatically generate the documentation associated with the code.
- **licence.txt:** contains a transcript of the license to which the code is subject.
- **ReadMe.md:** probably the file that brought you here.
- **report.pdf:** it is this same file and contains an account of what has been done.
- **report_benchmark.zip**: it contains the raw data, the processed data and the plots present in the report.
- **requirements.txt:** contains all the packages required by python to run the measurement script.

3.2 Build phase

First, you need to compile all the files. For this you need a GCC compiler, OpenMP, Doxygen and Graphviz installed on your machine. If not, the process may fail.

1. Head to the main directory.
2. Create a new build folder inside:
```
user@PC:~/dir$ mkdir build
```
3. Go to the build folder:
```
user@PC:~/dir$ cd build
```
4. Launch the build command with the parameter the root directory containing the CMakeLists.txt file:
```
user@PC:~/dir/build$ cmake ..
```
5. Then run the make command
```
user@PC:~/dir/build$ make all
```

At this point, in the build folder there will be all the compiled executables and in the main directory there will be a new folder named docs containing the source code documentation in html and latex.

3.3 Test phase

You can verify the correct functioning of the application provided by using the cmake package.

6. Go to the build folder (if you are following all the directions then you will already be in this one).
7. Launch the ctest command (or also make test):
```
user@PC:~/dir/build$ ctest
```

The desired output is as follows

```
100% tests passed, 0 tests failed out of 2
```

3.4 Install Python3 requirements phase

You need to check whether the python requirements described in the requirements.txt file are satisfied or not. If they are not, the authors recommend using a virtual environment to not occupy memory and override the python installation. This can be done as:

8. Head to the main directory.
9. Create the virtual environment from the current python3 installation (we used python3.8). If you don't have it, you need the package python3.8-venv (installable with the command `sudo apt install python3.8-venv`).
```
user@PC:~/dir$ python3 -m venv .venv
```
10. Now you need to install all the required packages:
```
user@PC:~/dir$ source ./.venv/bin/activate
(.venv) user@PC:~/dir$ pip3 install -r requirements.txt
(.venv) user@PC:~/dir$ deactivate
```

3.5 Measure phase

At this point, everything is ready to run the measurement script. To run the script, you need to run the following commands

```
user@PC:~/dir$ bash script/measures.bash
```

You can see the output in the measurements folder as indicated in 3.1, but just the raw data. To extract the processed data and the plots you can use the command:

```
(.venv) user@PC:~/dir$ python3 ./script/extract_measures.py
```

Note: Once you have performed steps 3.2 to 3.4, you no longer need to perform them. Step 3.5 can be performed directly to obtain new measurements. In case of changes to the source code it will be necessary to repeat steps 3.2 and 3.3.

3.5.1 Change measurement parameters

it is likely that, depending on the machine on which you want to perform the measurement, you want to edit the measurement parameters such as the dimensions tested, the number of threads used, the optimizations used and the number of measurements for each configuration.

To do this, you need to go to the */script/measures.bash* and in the */script/extract_measures.py* file. Immediately after the license and the fields intended for the file's metadata, some constants are defined as shown here

```
ARRAY_RC=[1000, 10000, 100000, 1000000, 10000000]
ARRAY_RANGE=[1000, 10000, 100000]
ARRAY_THS=[0, 1, 2, 4, 8]
ARRAY_OPT=[0, 1, 2, 3]
```

and here

```
NMEASURES=100

ARRAY_RC=(1000 10000 100000 1000000 10000000)
ARRAY_RANGE=(1000 10000 100000)
ARRAY_THS=(0 1 2 4 8)
ARRAY_OPT=(0 1 2 3)
```

The configurations are given by all the possible combination *(ARRAY_RC, ARRAY_RANGE, ARRAY_THS, ARRAY_OPT)*. Therefore, if these have lengths 5, 3, 5 and 3 respectively, $3 \cdot 3 \cdot 5 \cdot 3 = 135$ configurations will be obtained.

- To change the configurations simply add or remove values from this fields (in the *ARRAY_RC* 0 **must** be present to correctly evaluate the speedup and the efficiency).
- To change the number of repetitions for each configuration just change the NMEASURES value.

Warning: the two configurations in the two files **must** match.

4 Experimental Setup

4.1 OS setup

All measurements were made using the Windows 10 operating system. Ubuntu 20.04 LTS virtualized using WSL 2 was used. We were using gcc9 as compiler

4.2 Hardware setup

The hardware configuration is reported at the time of measuring the performance of the solutions provided. The configuration is extracted from `/proc/cpuinfo` and `/proc/meminfo`.

```
user@PC:~/dir$ cat /proc/cpuinfo
(…omitted firt 7 threads…)
processor       : 7
vendor_id       : AuthenticAMD
cpu family      : 23
model           : 17
model name      : AMD Ryzen 5 2500U with Radeon Vega Mobile Gfx
stepping        : 0
microcode       : 0xffffffff
cpu MHz         : 1996.185
cache size      : 512 KB
physical id     : 0
siblings        : 8
core id         : 3
cpu cores       : 4
apicid          : 7
initial apicid  : 7
fpu             : yes
fpu_exception   : yes
cpuid level     : 13
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 ht syscall nx mmxext fxsr_opt pdpe1gb rdtscp lm constant_tsc rep_good
nopl tsc_reliable nonstop_tsc cpuid extd_apicid pni pclmulqdq ssse3 fma cx16 sse4_1 sse4_2
movbe popcnt aes xsave avx f16c rdrand hypervisor lahf_lm cmp_legacy cr8_legacy abm sse4a
misalignsse 3dnowprefetch osvw topoext ssbd ibpb vmmcall fsgsbase bmi1 avx2 smep bmi2 rdseed
adx smap clflushopt sha_ni xsaveopt xsavec xgetbv1 xsaves clzero xsaveerptr virt_ssbd arat
bugs            : sysret_ss_attrs null_seg spectre_v1 spectre_v2 spec_store_bypass
bogomips        : 3992.37
TLB size        : 2560 4K pages
clflush size    : 64
cache_alignment : 64
address sizes   : 48 bits physical, 48 bits virtual
power management:
```

```
user@PC:~/dir$ cat /proc/meminfo
MemTotal:        5587448 kB
MemFree:         4452404 kB
MemAvailable:    4709292 kB
Buffers:           27012 kB
Cached:           417884 kB
SwapCached:            0 kB
Active:           207008 kB
Inactive:         783184 kB
Active(anon):        144 kB
Inactive(anon):   545212 kB
Active(file):     206864 kB
Inactive(file):   237972 kB
Unevictable:           0 kB
Mlocked:               0 kB
SwapTotal:       2097152 kB
SwapFree:        2097152 kB
```

```
Dirty:                12 kB
Writeback:             0 kB
AnonPages:        545324 kB
Mapped:            91368 kB
Shmem:                72 kB
KReclaimable:      37652 kB
Slab:              66744 kB
SReclaimable:      37652 kB
SUnreclaim:        29092 kB
KernelStack:        3472 kB
PageTables:        15132 kB
NFS_Unstable:          0 kB
Bounce:                0 kB
WritebackTmp:          0 kB
CommitLimit:     4890876 kB
Committed_AS:     756860 kB
VmallocTotal:   34359738367 kB
VmallocUsed:       23056 kB
VmallocChunk:          0 kB
Percpu:             2432 kB
AnonHugePages:    182272 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:      101376 kB
DirectMap2M:     3590144 kB
DirectMap1G:     2097152 kB
```

Additionally, the available physical memory is reported:

```
user@PC:~/dir$ echo $(($(getconf _PHYS_PAGES) * $(getconf PAGE_SIZE) / (1024 * 1024))) "MB"
5456 MB
```

Finally, the available virtual memory is reported:

```
user@PC:~/dir$ echo $(($(getconf _AVPHYS_PAGES) * $(getconf PAGE_SIZE) / (1024 * 1024))) "MB"
4348 MB
```

5 Results

5.1 Results organization

Measurements are made on different dimensions of the problem, using a sequential version of the algorithms and a parallelized version by trying with different numbers of active threads. All the possibilities analyzed are divided by size of the problem.

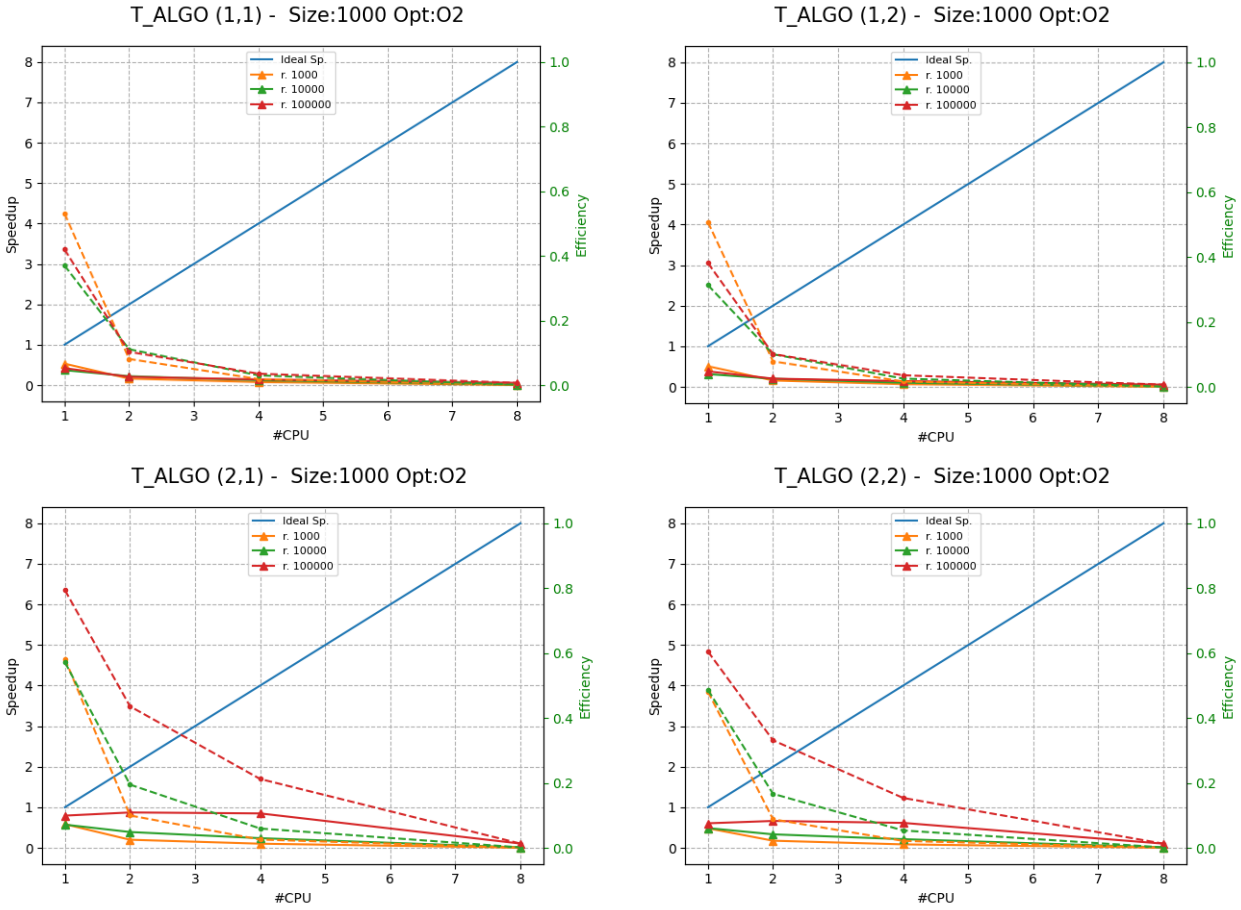
Each box consisting of 4 images contains all the combinations of proposed algorithms. Each image contains the ideal speedup (blue line), experimental speedup (continuous broken lines), and efficiency (broken lines).

In this section of the report, only O2-optimized results are commented on because they give the best results. The others have however been provided and have the same meaning (except for performance improvements due to optimization).

The raw, processed data and the plots of these were provided for the purpose of consultation. The tabulated data from which the plots are generated are not reported for the sake of brevity but are found within the project.

5.2 Optimization O2

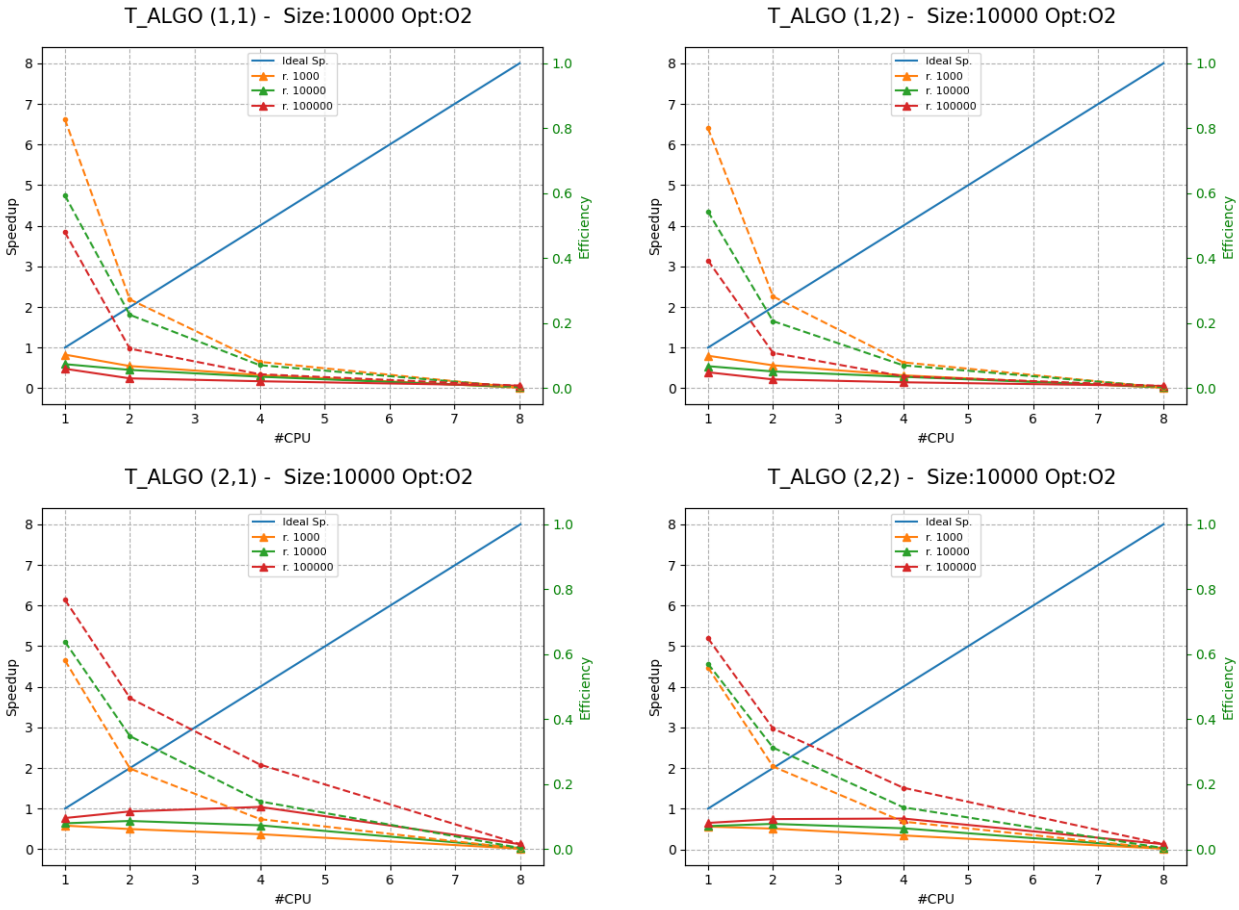
5.2.1 Size 10^3



As we can see from the results, with this size of the array to be sorted it is not at all convenient to parallelize the algorithm in any case. This is because in none of the three steps the size of the problem is such as to amortize the overhead introduced by OpenMP.

- **1-1, 1-2:** the overhead is very high as extremely large arrays are allocated compared to the size of the problem (in the phase of counting the occurrences), one for each thread, bringing the speedup almost to 0.
- **2-1, 2-2:** a better performance is noted with a very wide range of values compared to the dimension ($k \gg n$) because the PMF vector is scattered and therefore it is possible to effectively parallelize the execution. Unfortunately, the domain of application of this algorithm is the opposite one ($k \ll n$) so this behavior cannot be used.

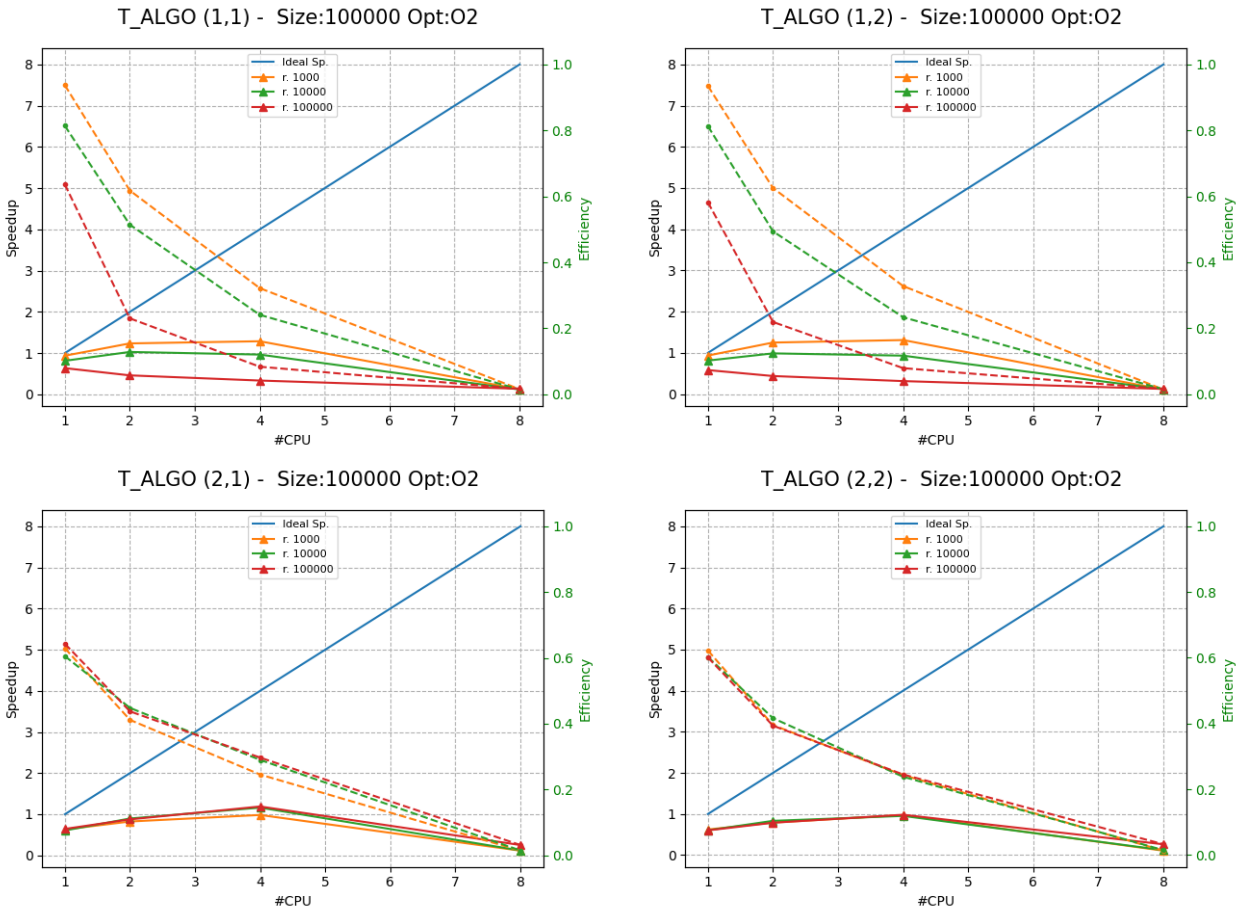
5.2.2 Size 10^4



The results for this dimension of the problem with the ranges of values analyzed are similar. It can be understood that increasing the size of the problem in all the algorithms is reducing the overhead, probably due to the minimum and maximum algorithm which, as the size of the array increases, justifies the overhead introduced by the generation of more threads.

In any case there is no performance improvement over the sequential algorithm, so it makes no sense to use the parallelized algorithm on this dimension.

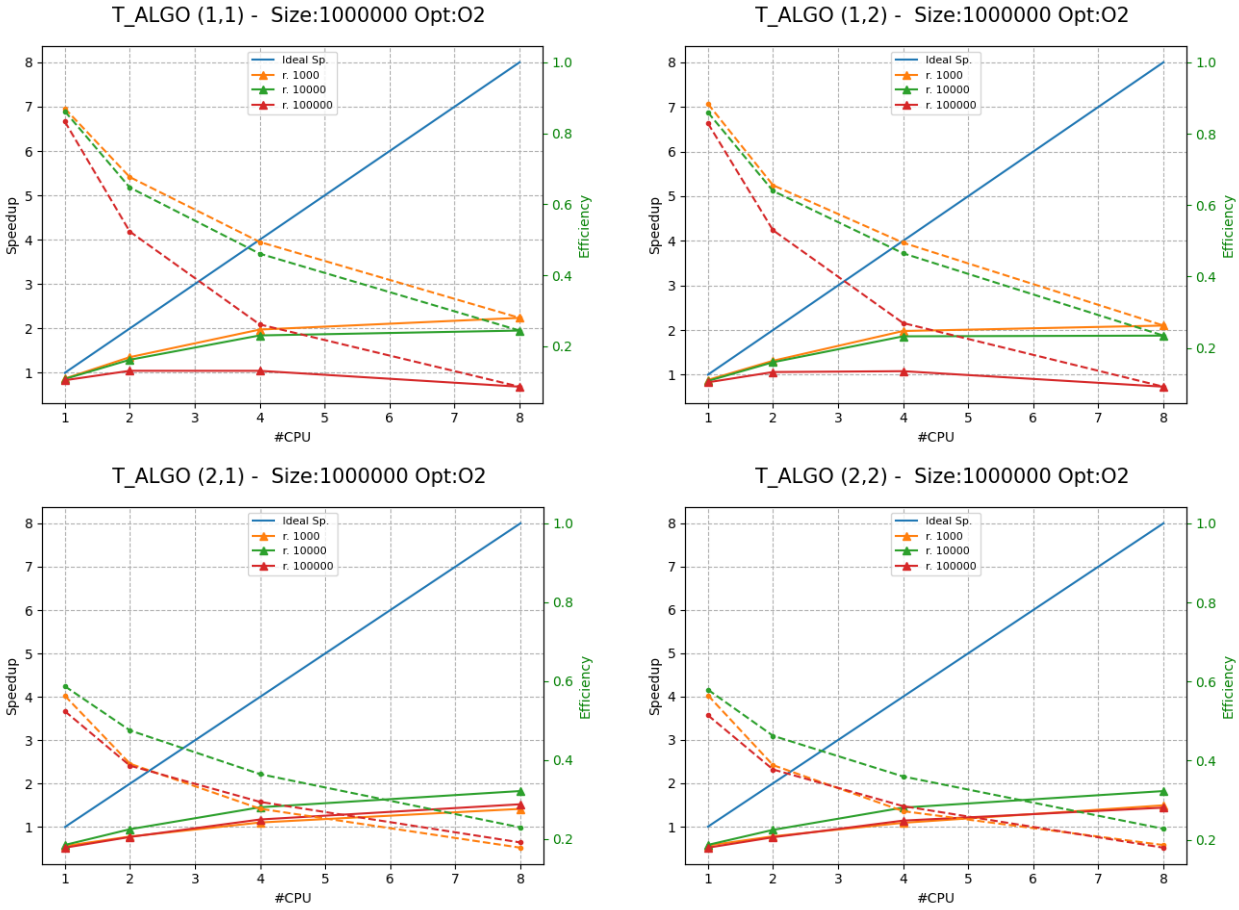
5.2.3 Size 10^5



With this dimension of the problem the improvement can be seen but it is not enough to justify the use of the algorithm.

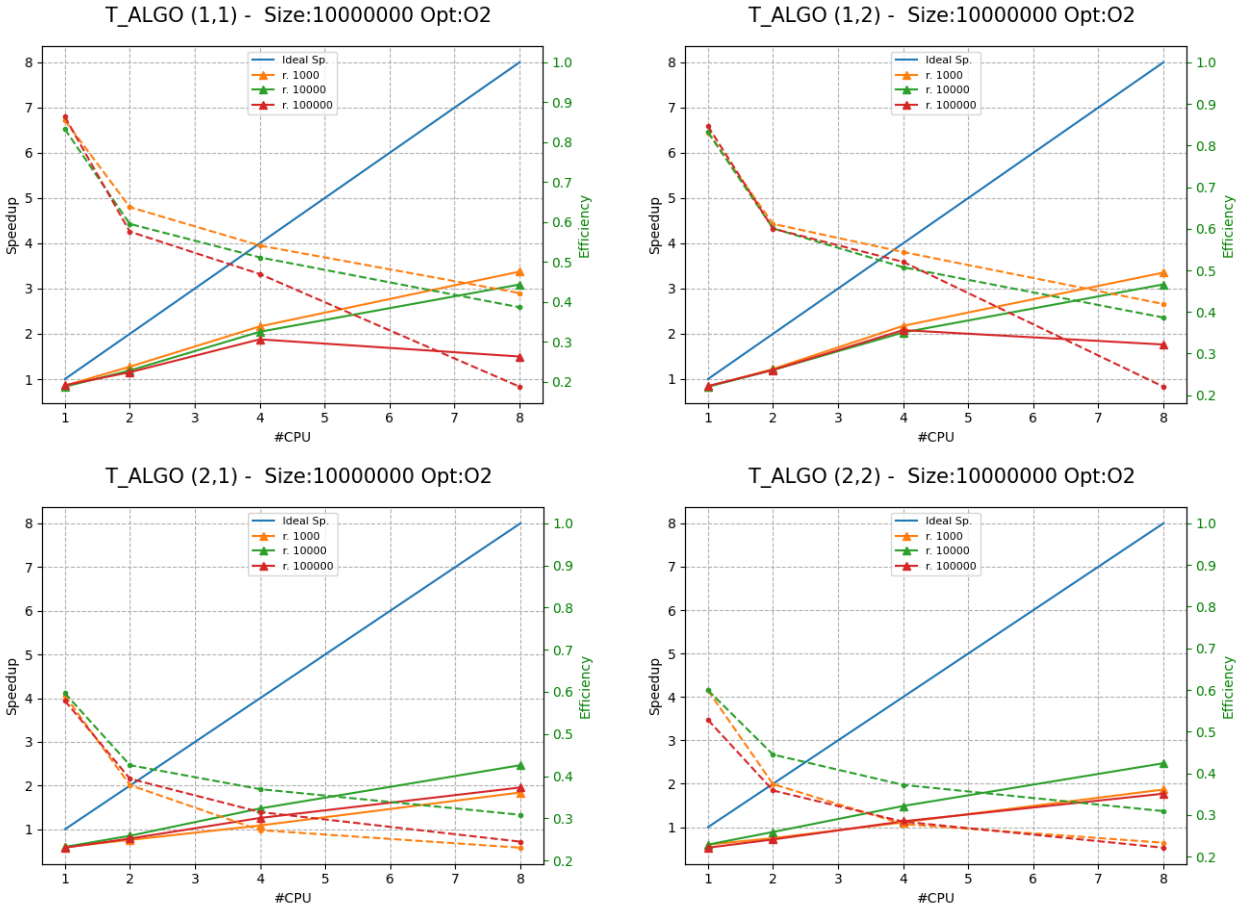
- **1-1, 1-2**: When n is greater than k by three orders of magnitude, improvements begin to appear, since the allocation of a vector of size k for each thread to count the occurrences and the relative reduction phase begin to be amortized. Moreover, they probably start to increase the performances of the maximum and minimum and population algorithms (although without introducing a very high speed up, just over 1 for the first 4 threads).
- **2-1, 2-2**: It can be noted that as n increases with respect to k, the performances decrease due to the atomic operation which cannot be performed in parallel. In any case, the benefits go up as the maximum and minimum algorithms and the population algorithm begin to be at least amortized.

5.2.4 Size 10^6



- **1-1, 1-2:** With n even greater than k it can be understood that the occurrence count algorithm starts to work, as well as the other two algorithms, starting to produce almost usable results with 2 and 4 threads, reaching a speedup of 1.35 and 1.98 with respective efficiency of 68% and 49% in cases where the range has dimensions 10^3. It can also be noted that since the results are almost the same, the parallelization of the CDF array calculation starting from that of the PDF does not seem to introduce benefits due to the high dependence between successive data.
- **2-1, 2-2:** In this case we notice that the speedup tends to go up even if by very little and unacceptable efficiency values. However, it confirms that the maximum and minimum and population algorithms tend to work with this dimension of the problem. The count of occurrences instead introduces overhead due to threads that fail to perform the atomic operation in a truly parallel way (as the values start to become sparse).

5.2.5 Size 10^7

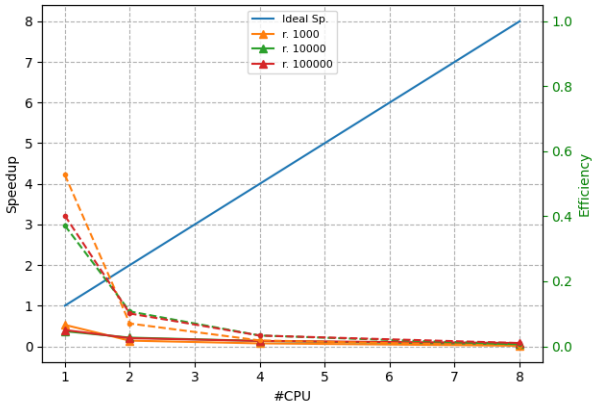


- **1-1, 1-2:** With these dimensions, even executions with a larger range have better timings (confirming what has been said previously). Furthermore, the solution with a smaller range becomes even better to the point of justifying the use of 8 threads (obviously not with the best efficiency). The hypothesis is also confirmed that the parallelization of the passage from the vector of the PMF to that of the CDF does not introduce benefits.
- **2-1, 2-2**: Regarding this solution, again for the reasons described above, it does not achieve good results.

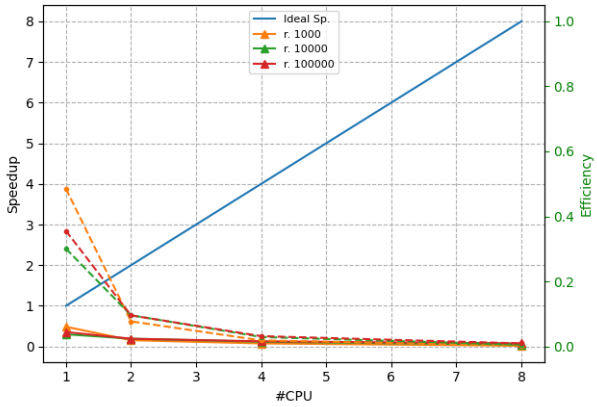
5.3 Optimization O0

5.3.1 Size 10^3

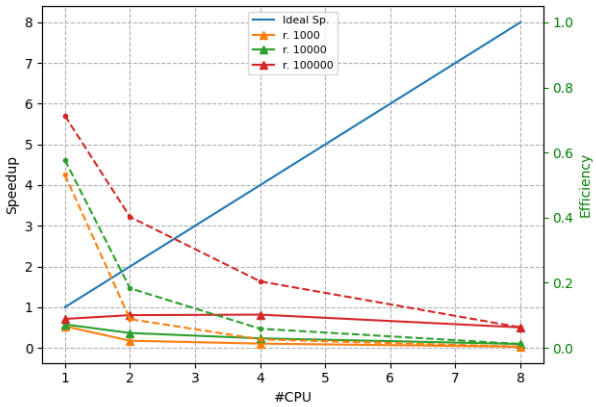
T_ALGO (1,1) - Size:1000 Opt:O0



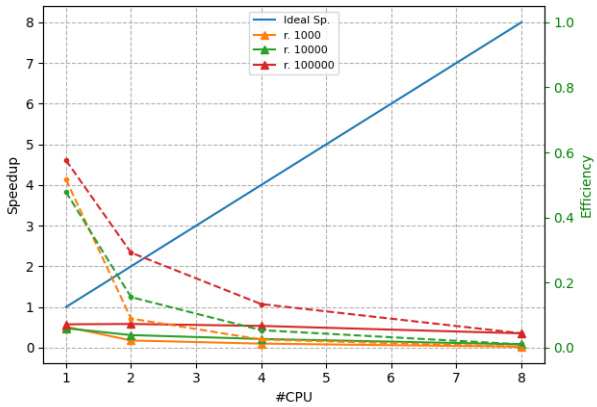
T_ALGO (1,2) - Size:1000 Opt:O0



T_ALGO (2,1) - Size:1000 Opt:O0

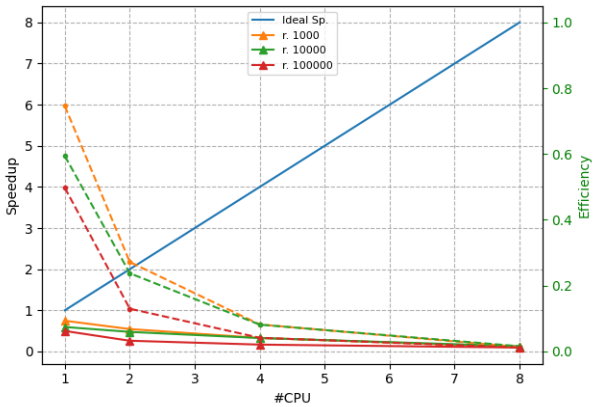


T_ALGO (2,2) - Size:1000 Opt:O0

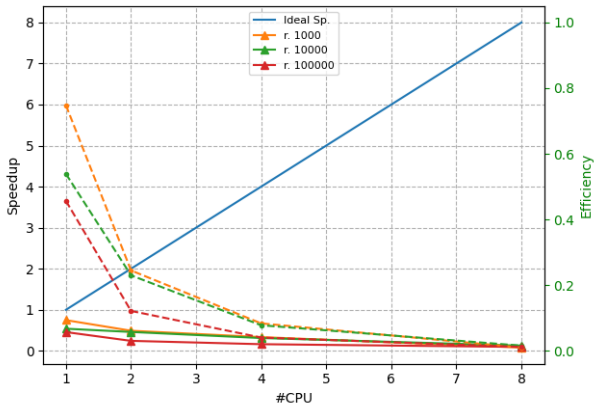


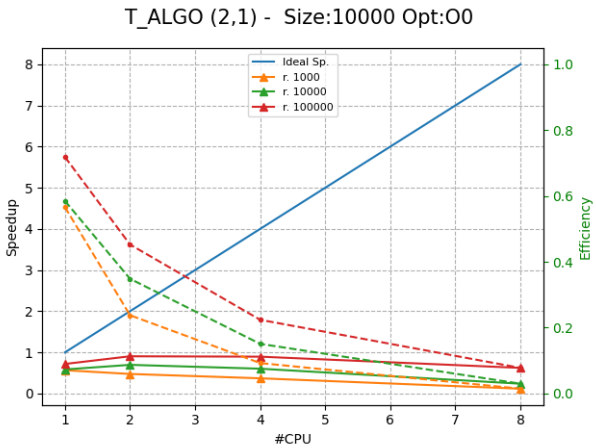
5.3.2 Size 10^4

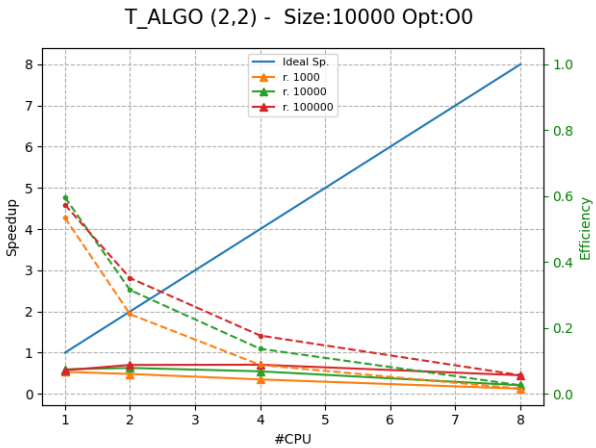
T_ALGO (1,1) - Size:10000 Opt:O0



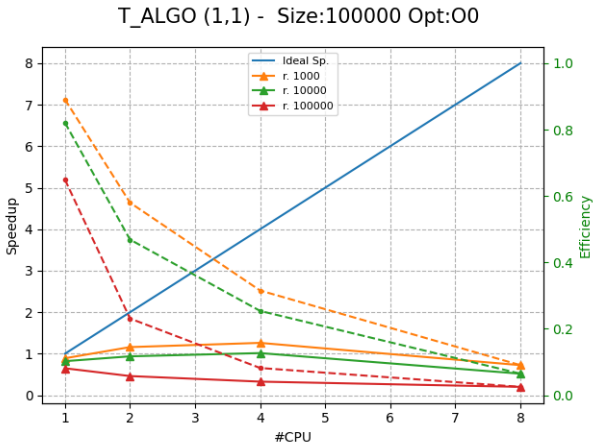
T_ALGO (1,2) - Size:10000 Opt:O0

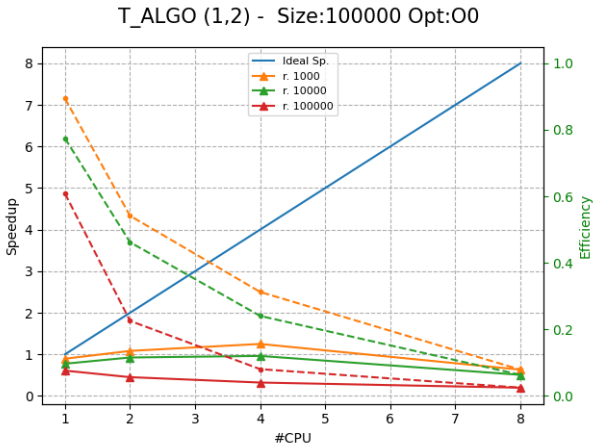


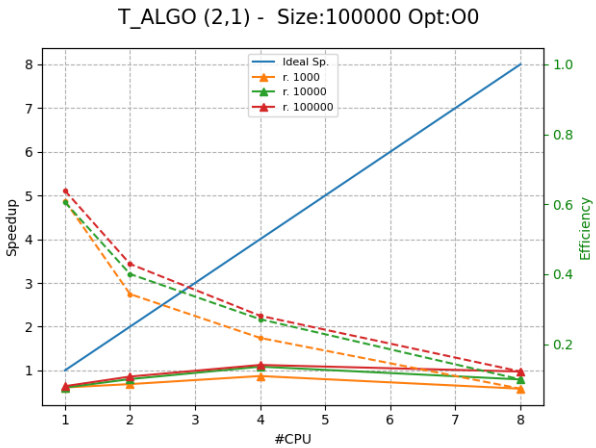

T_ALGO (2,1) - Size:10000 Opt:O0
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

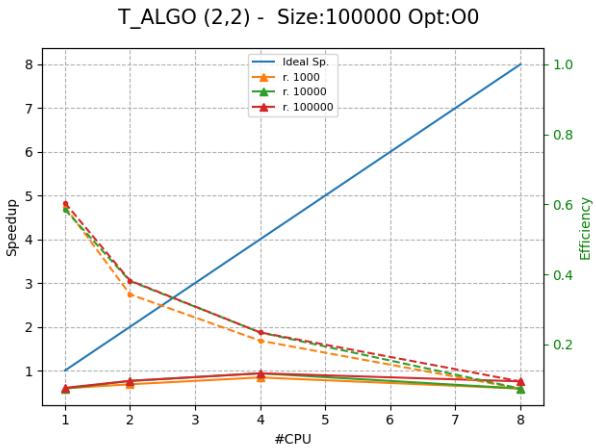

T_ALGO (2,2) - Size:10000 Opt:O0
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

5.3.3 Size 10^5


T_ALGO (1,1) - Size:100000 Opt:O0
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

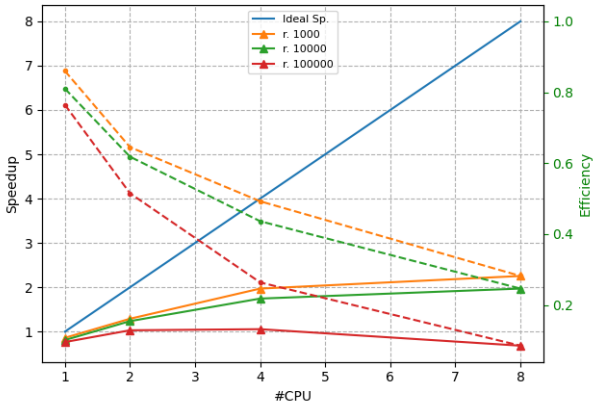

T_ALGO (1,2) - Size:100000 Opt:O0
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU


T_ALGO (2,1) - Size:100000 Opt:O0
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

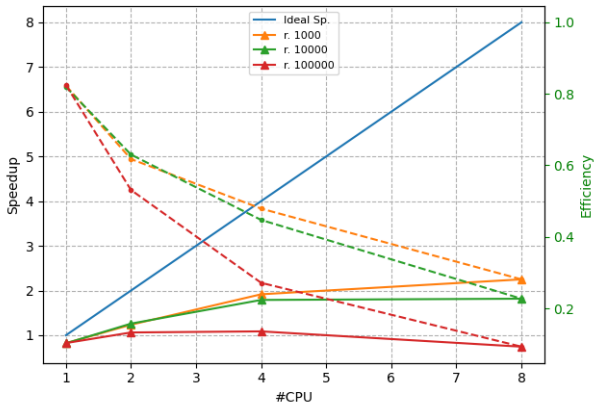

T_ALGO (2,2) - Size:100000 Opt:O0
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

5.3.4 Size 10^6

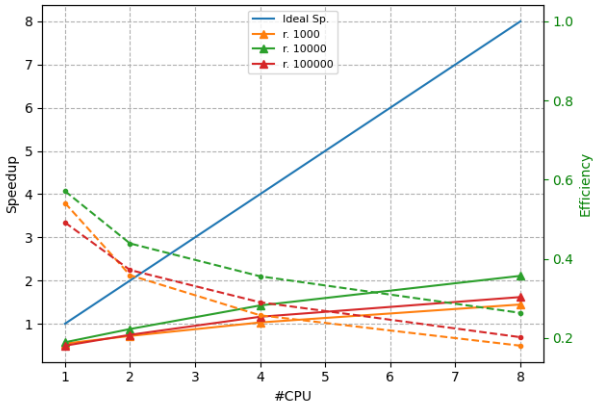
T_ALGO (1,1) - Size:1000000 Opt:O0



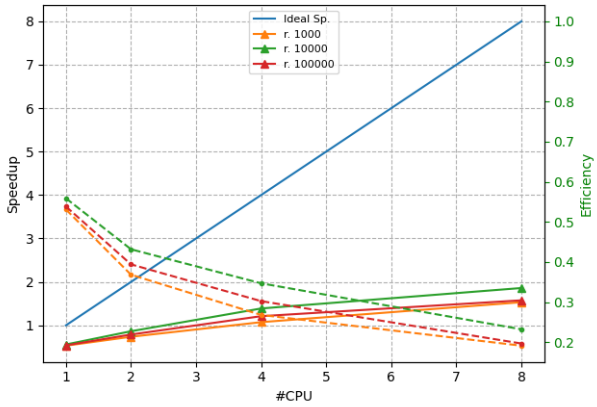
T_ALGO (1,2) - Size:1000000 Opt:O0



T_ALGO (2,1) - Size:1000000 Opt:O0

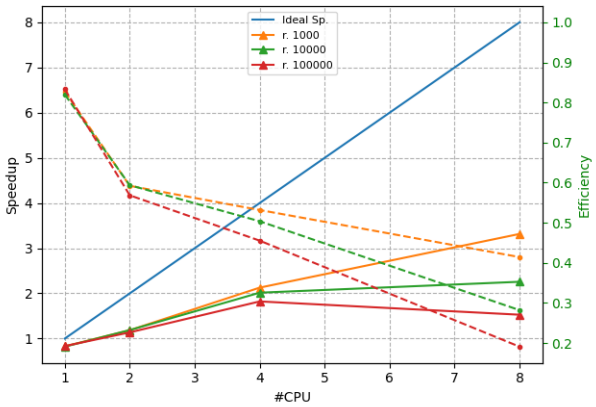


T_ALGO (2,2) - Size:1000000 Opt:O0

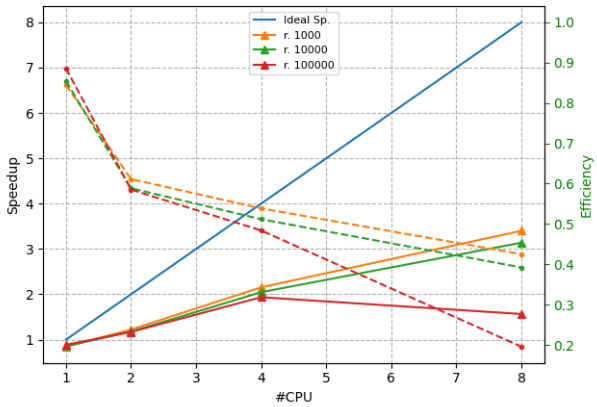


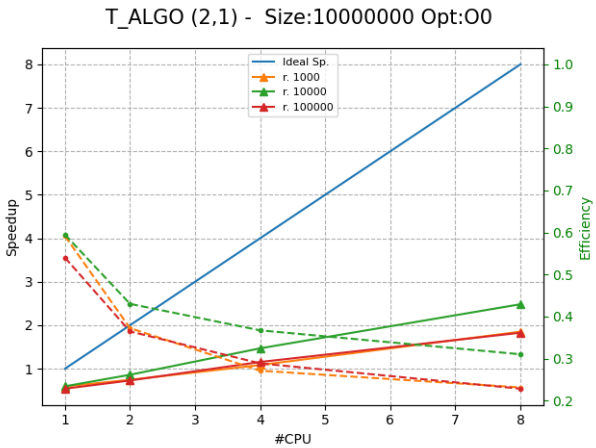
5.3.5 Size 10^7

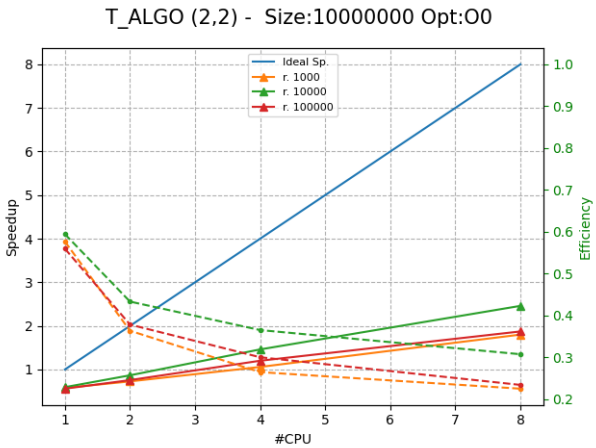
T_ALGO (1,1) - Size:10000000 Opt:O0



T_ALGO (1,2) - Size:10000000 Opt:O0

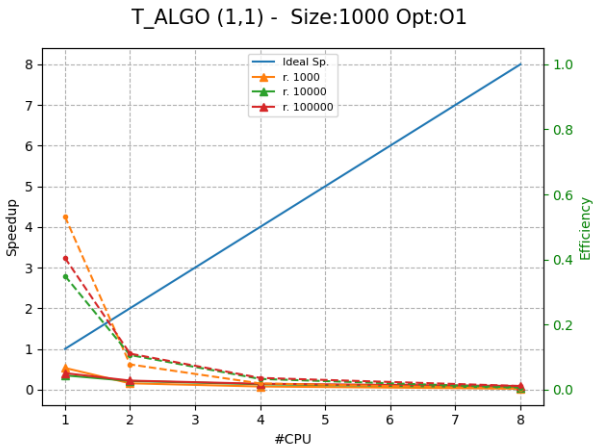


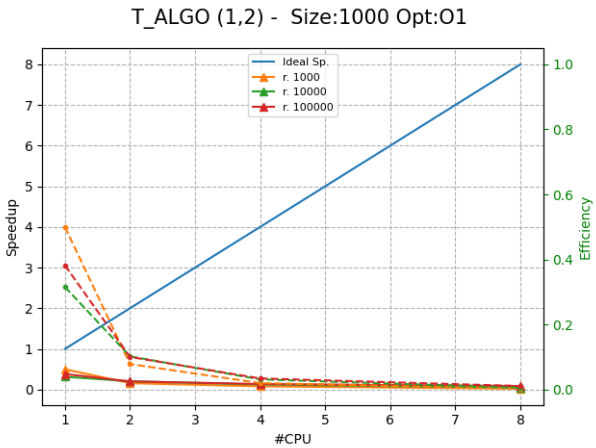


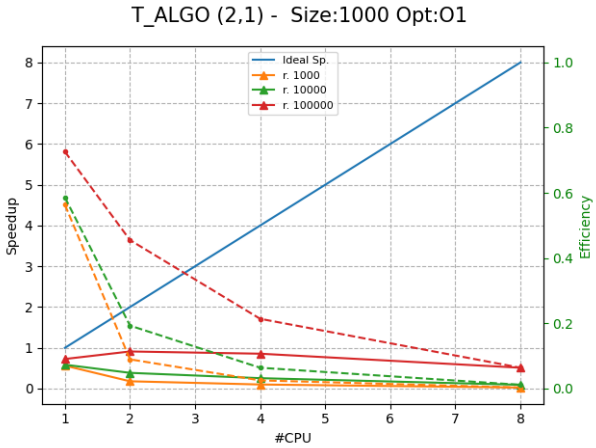


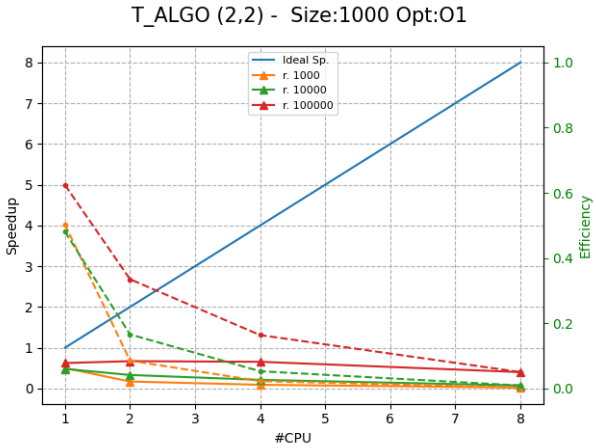
5.4 Optimization O1

5.4.1 Size 10^3

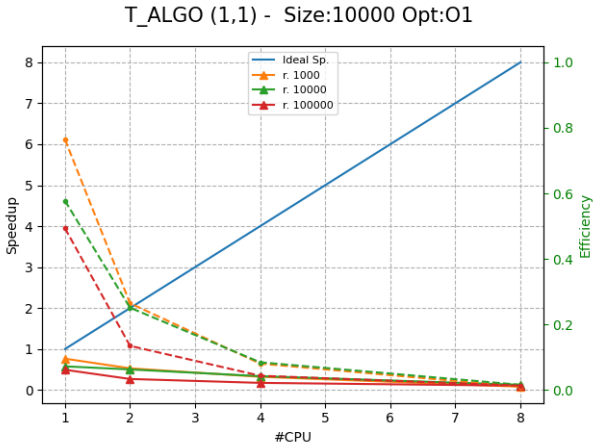


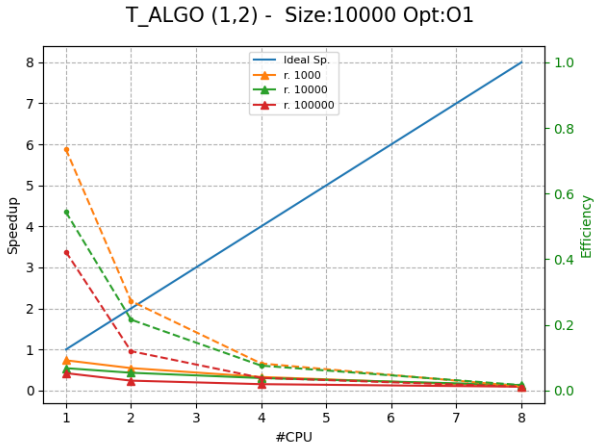


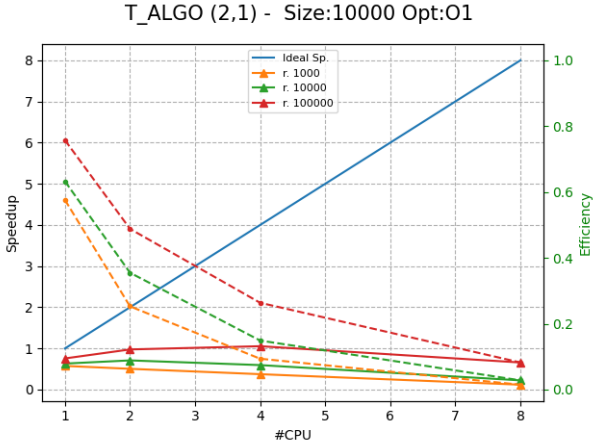


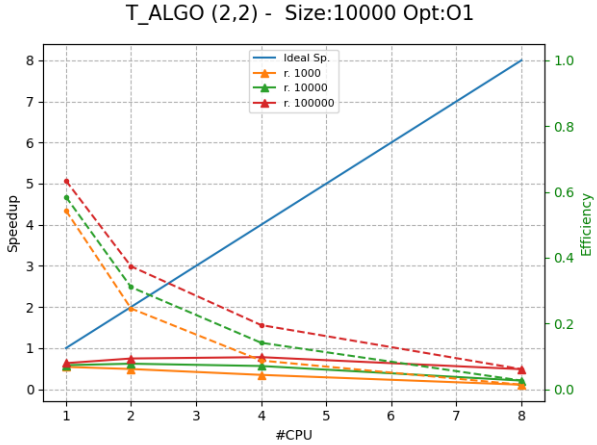


5.4.2 Size 10^4

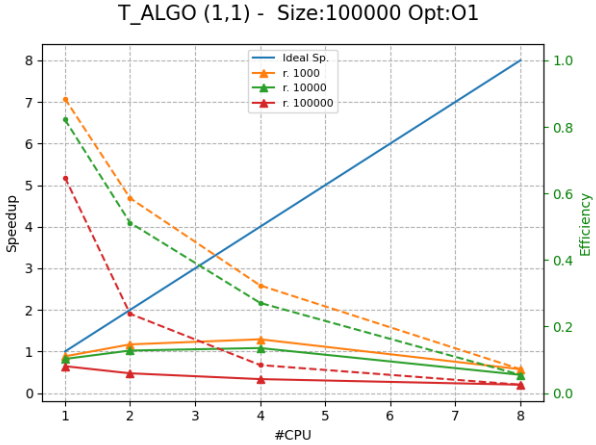


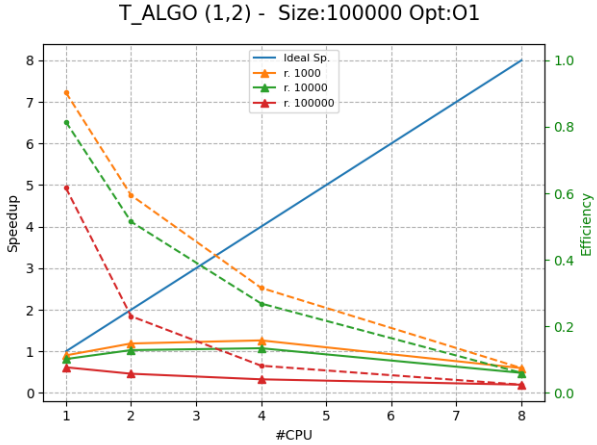


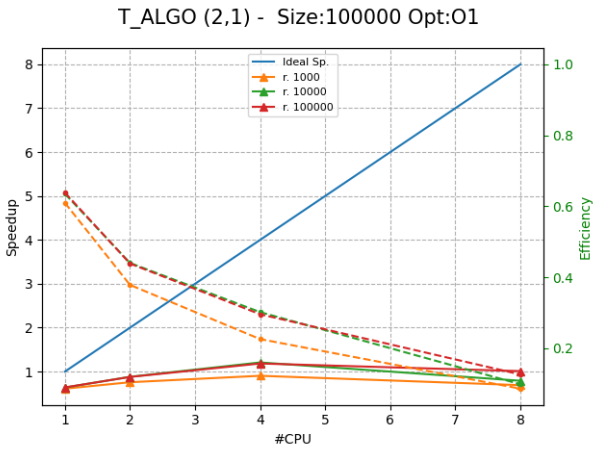


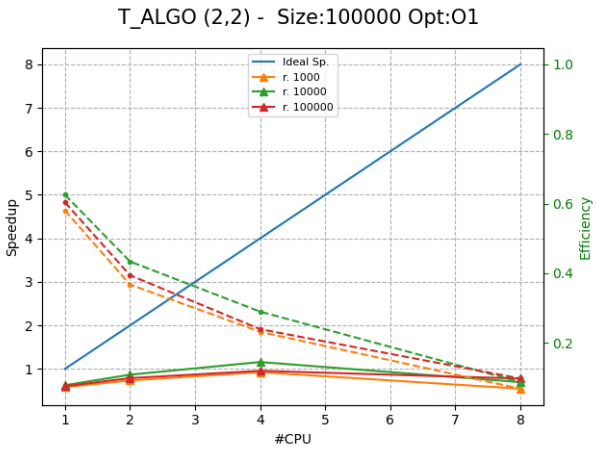


5.4.3 Size 10^5

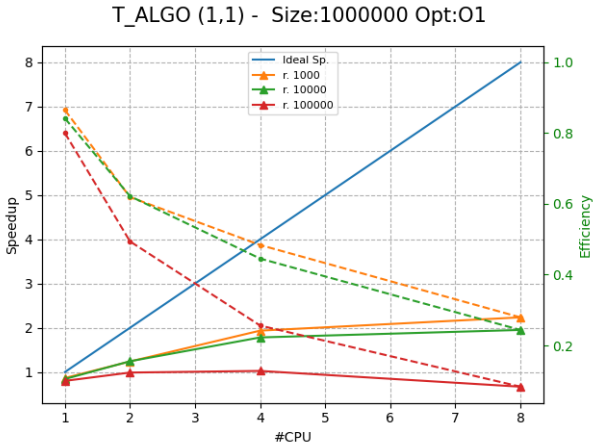


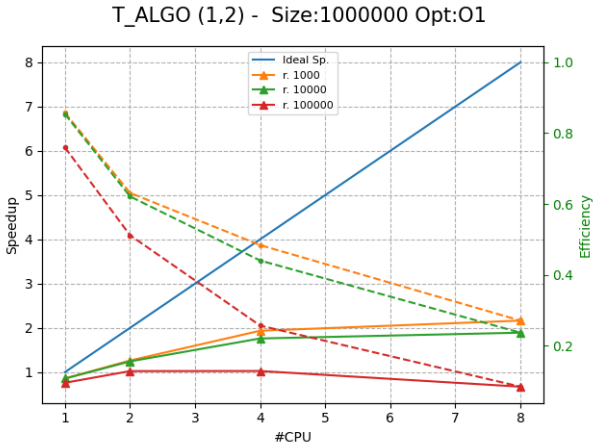


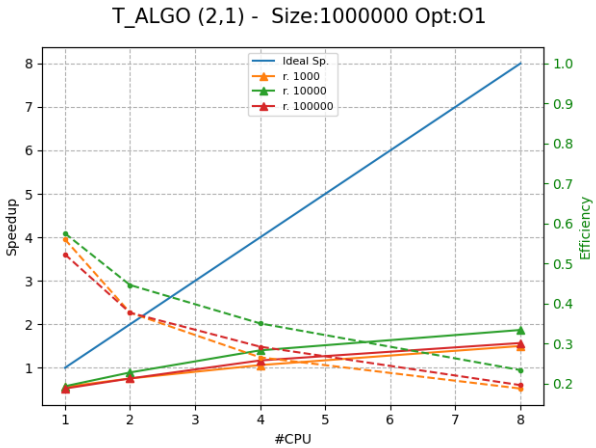

T_ALGO (2,1) - Size:100000 Opt:O1
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

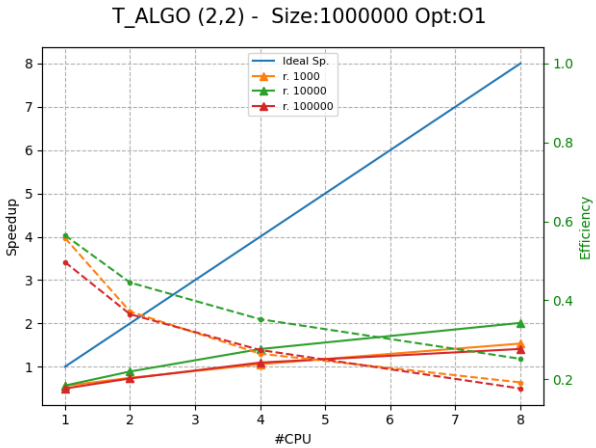

T_ALGO (2,2) - Size:100000 Opt:O1
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

5.4.4 Size 10^6


T_ALGO (1,1) - Size:1000000 Opt:O1
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

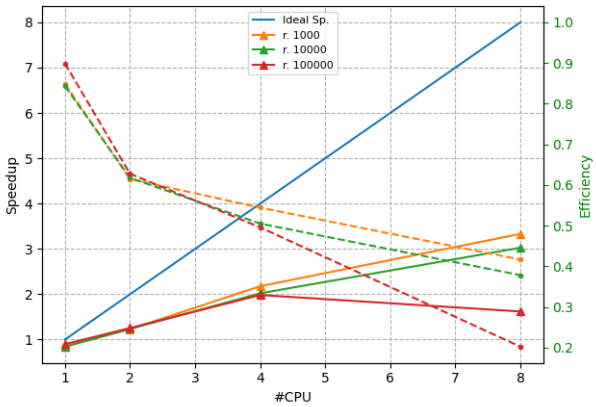

T_ALGO (1,2) - Size:1000000 Opt:O1
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU


T_ALGO (2,1) - Size:1000000 Opt:O1
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

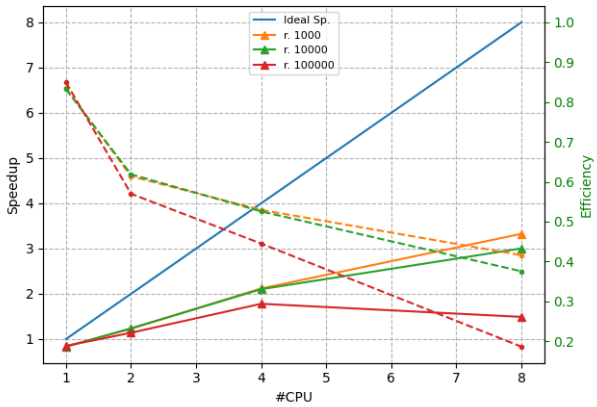

T_ALGO (2,2) - Size:1000000 Opt:O1
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

5.4.5 Size 10^7

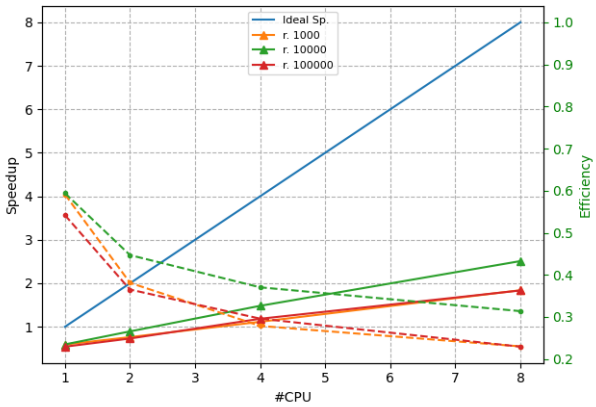
T_ALGO (1,1) - Size:10000000 Opt:O1



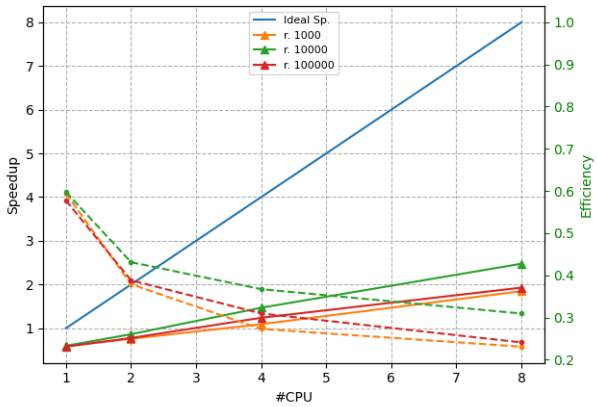
T_ALGO (1,2) - Size:10000000 Opt:O1



T_ALGO (2,1) - Size:10000000 Opt:O1



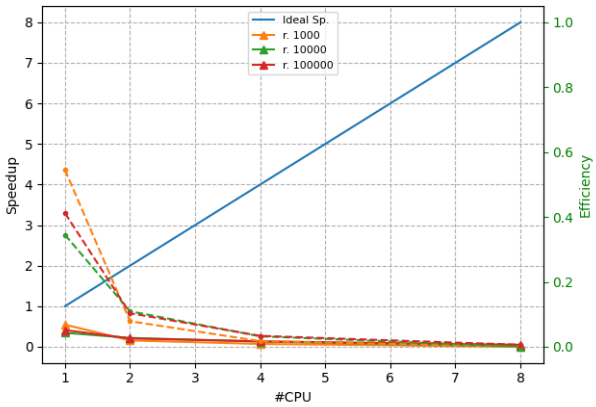
T_ALGO (2,2) - Size:10000000 Opt:O1



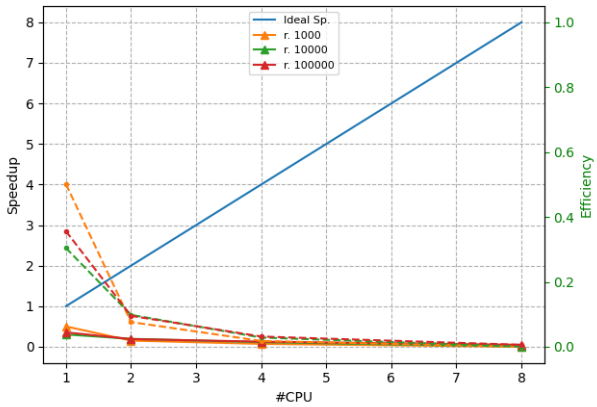
5.5 Optimization O3

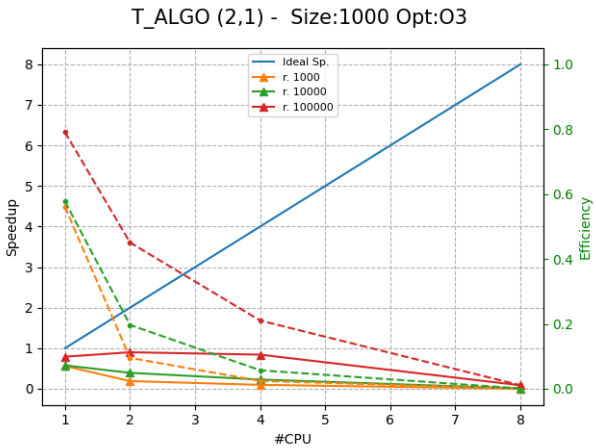
5.5.1 Size 10^3

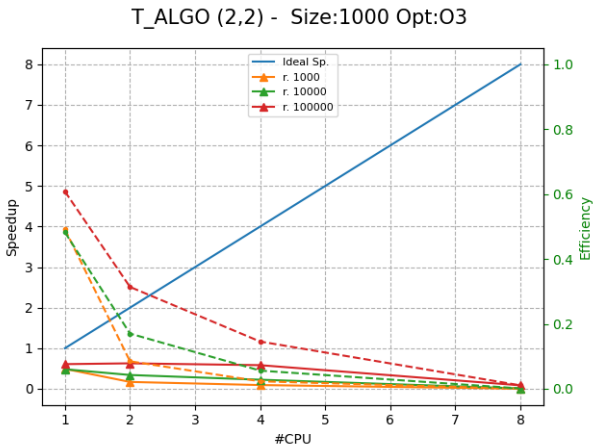
T_ALGO (1,1) - Size:1000 Opt:O3



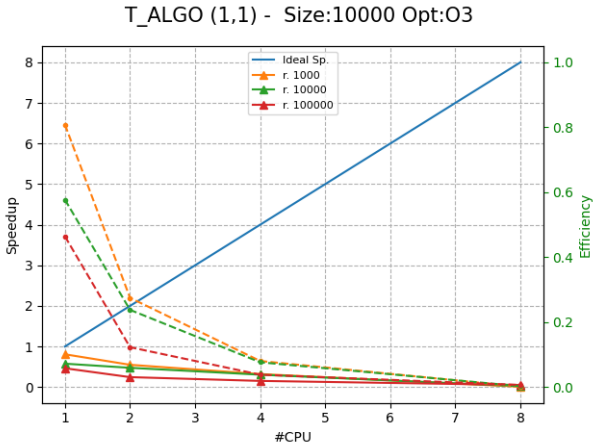
T_ALGO (1,2) - Size:1000 Opt:O3

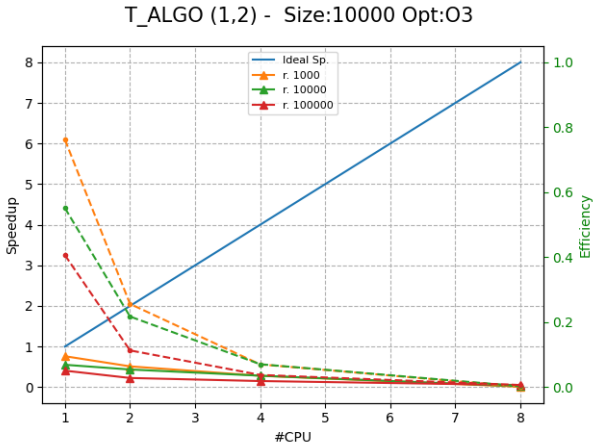


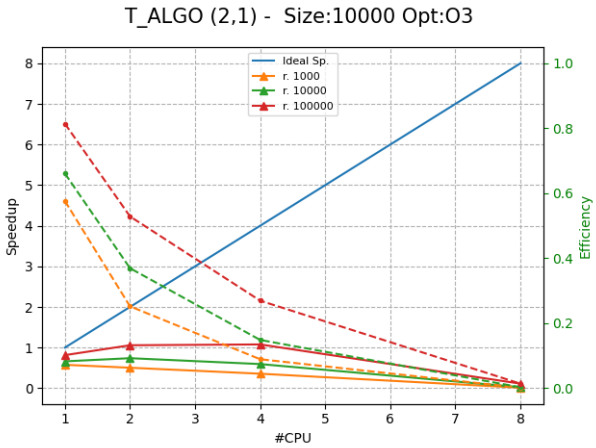

T_ALGO (2,1) - Size:1000 Opt:O3
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

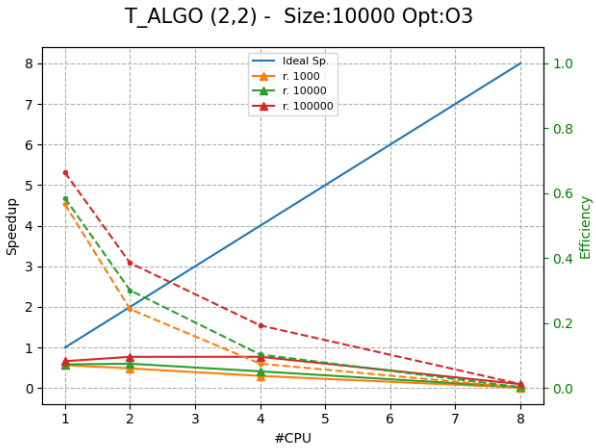

T_ALGO (2,2) - Size:1000 Opt:O3
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

5.5.2 Size 10^4


T_ALGO (1,1) - Size:10000 Opt:O3
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

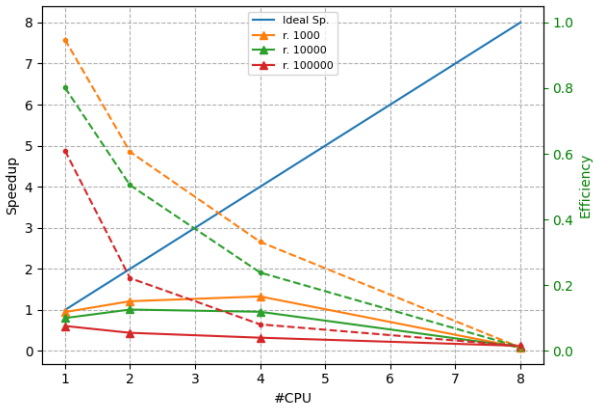

T_ALGO (1,2) - Size:10000 Opt:O3
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU


T_ALGO (2,1) - Size:10000 Opt:O3
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

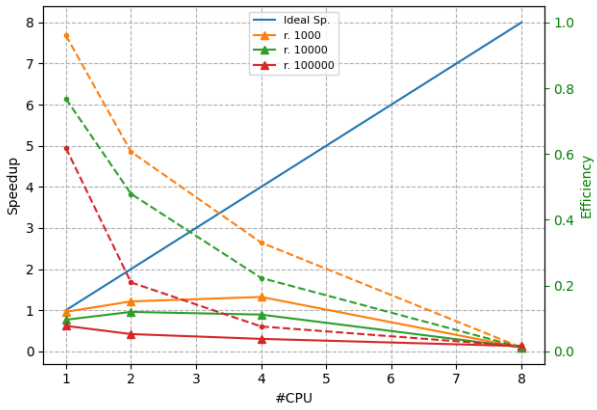

T_ALGO (2,2) - Size:10000 Opt:O3
Ideal Sp.
r. 1000
r. 10000
r. 100000
Speedup
Efficiency
#CPU

5.5.3 Size 10^5

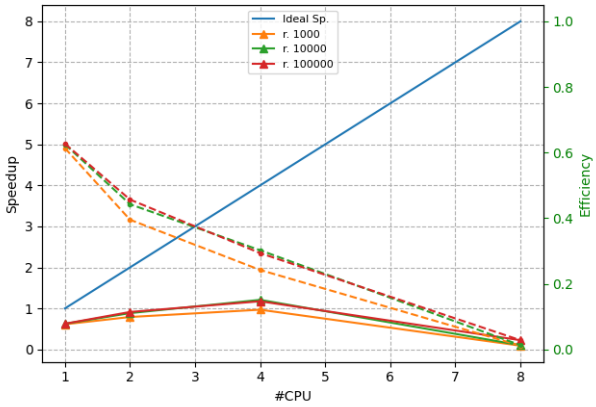
T_ALGO (1,1) - Size:100000 Opt:O3



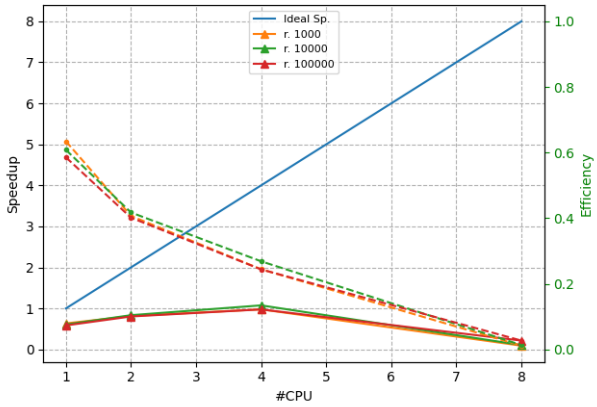
T_ALGO (1,2) - Size:100000 Opt:O3



T_ALGO (2,1) - Size:100000 Opt:O3

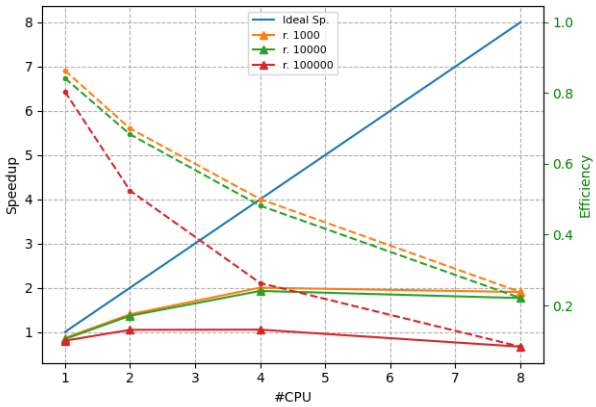


T_ALGO (2,2) - Size:100000 Opt:O3

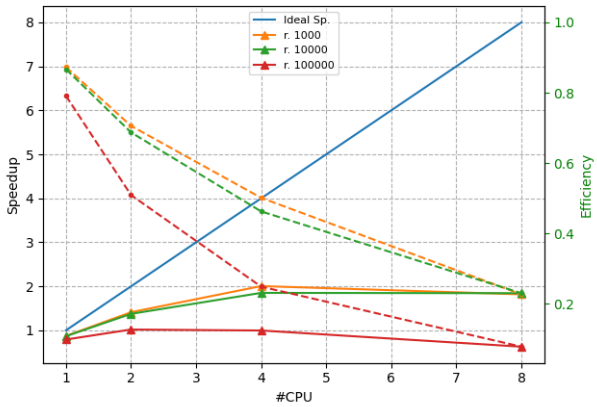


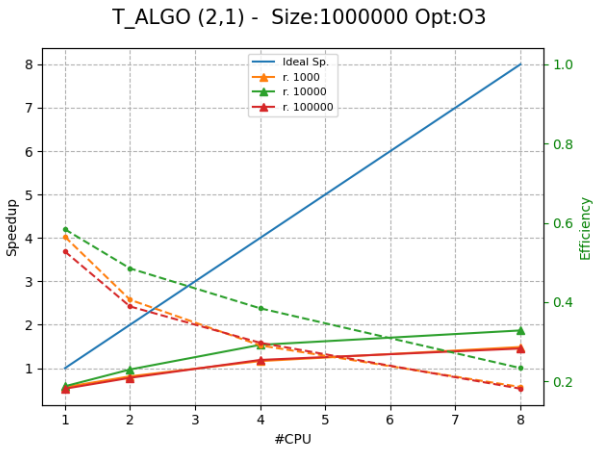
5.5.4 Size 10^6

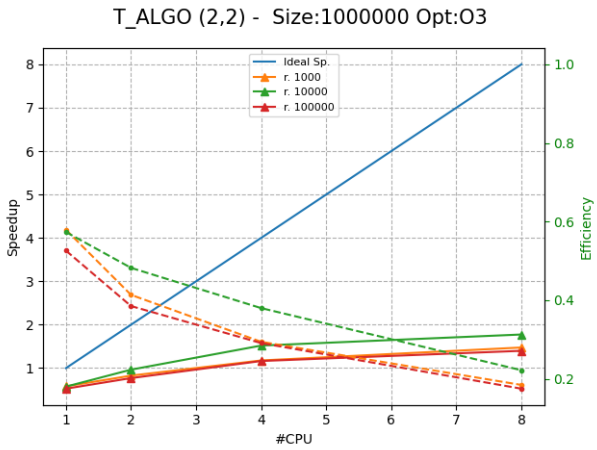
T_ALGO (1,1) - Size:1000000 Opt:O3



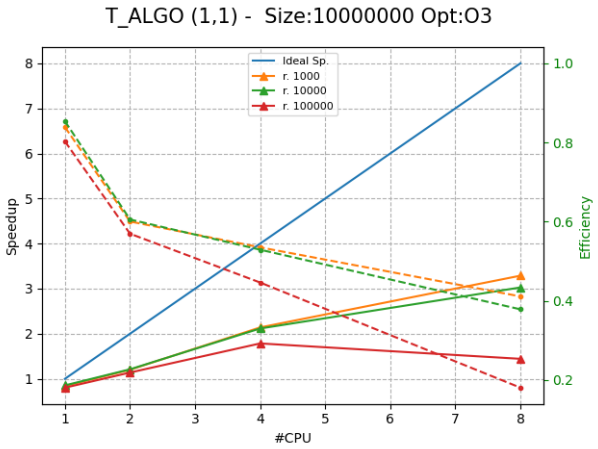
T_ALGO (1,2) - Size:1000000 Opt:O3

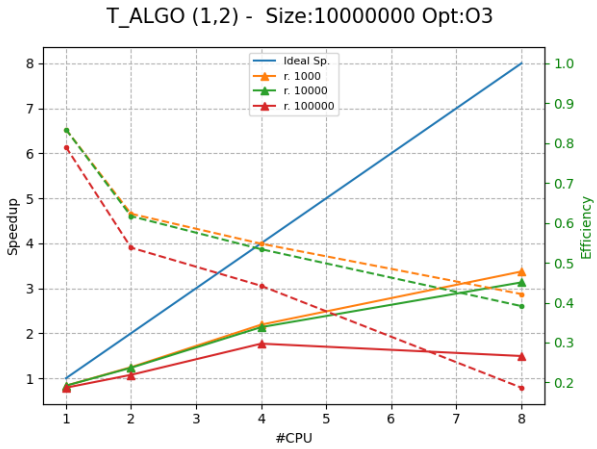


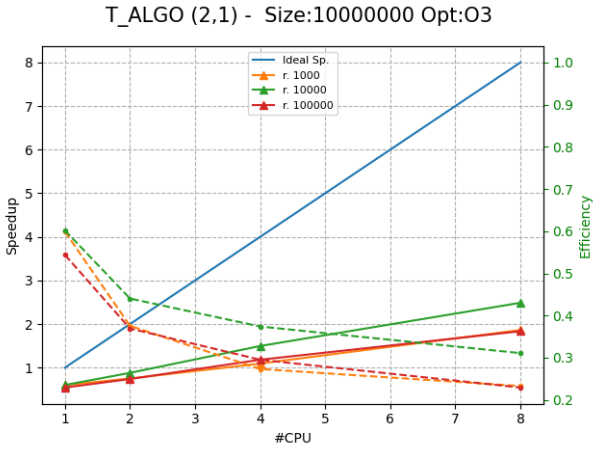


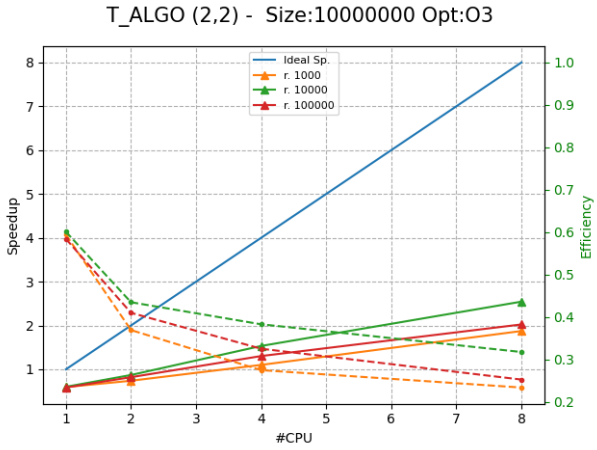


5.5.5 Size 10^7









5.6 Final considerations

The 1-1 solution is certainly the best of those tried. The best setup to use it (obviously compared to our configuration) is with 2 or 4 threads depending on the size of the array and the orders of magnitude of difference between n and k.

It is recommended to use 2 threads for $n \geq 10^6$ and k at least 3 orders of magnitude less than n, or 4 threads in the case in which $n \geq 10^7$ and $k \ll n$. For smaller sizes, the use of the parallelized version of the algorithm is not recommended.

6 API

The APIs provided in the files are generated at compile time. Here is a copy.

6.1 util.h

This library provides the functions to generate random arrays and a macro which, if defined, always generates the same arrays by fixing the seed. The STARTTIME and ENDTIME macros are also defined.

◆ init_rand_vector()

```
void init_rand_vector ( ELEMENT_TYPE ** A,
                        size_t          A_len,
                        long            min_value,
                        long            max_value
                      )
```

Initialize an array of a certain lenght with random integers in a certain range.

Parameters

A	The pointer to the pointer to the vector to initialize.
A_len	The desidered lenght of the array.
min_value	The minimum value inside the array.
max_value	The maximum value inside the array.

◆ deinit_rand_vector()

```
void deinit_rand_vector ( ELEMENT_TYPE * A )
```

Release the memory allocated for the array.

Parameters

A The pointer to the array to dealloc.

6.2 counting_sort.h

This library provides the functions to use the implemented counting sort algorithm, sequential and parallel, respectively.

◆ counting_sort()

```
void counting_sort ( ELEMENT_TYPE * A,
                     size_t         A_len
                   )
```

Reorder the array using a counter-sort alghorithm.

Parameters

A	The pointer to the array to reorder.
A_len	The lenght to the array to reorder.

◆ counting_sort_parall1()

```
void counting_sort_parall1 ( ELEMENT_TYPE * A,
                             size_t         A_len,
                             int            threads
                           )
```

Reorder the array using a counter-sort alghorithm parallelized.

Parameters

v	The pointer to the array to reorder.
A_len	The lenght to the array to reorder.
threads	The thread to use to run this function. If you want use default number use 0.

6.3 main_measures.c

This is the file used to launch the benchmarks you've seen in the previous sections.

7 Test Case

A total of 8 test cases were provided, 4 of which in the sequential version and 4 in the parallel version. Both run both compiled with OpenMP and not.

```
unsigned short empty_test();
unsigned short pempty_test();
unsigned short test1();
unsigned short test2();
unsigned short test3();
unsigned short ptest1();
unsigned short ptest2();
unsigned short ptest3();
```

- `empty_test` and `pempty_test` sort on an empty array. The expected result is that no errors occur.
- `test1` and `ptest1` perform a test on a random array and verify its ordering
- `test2` and `ptest2` perform a test where the ordered vector is known in advance and check the actual operation.
- `test` and `ptest3` perform the same test case as the previous ones, but on larger arrays.

Tests pass by all algorithm combinations at 100%.

License