Report
High Performance Computing

Implementation and Analysis of a parallel version of Prim's Algorithm

University of Salerno

High Performance Computing

A.Y 2023/2024

Alfonso Giso 0622701842 a.giso@studenti.unisa.it

Index

Problem Description

The goal of the project is to implement and analyze the parallel version of the Prim's Algorithm.
This algorithm deals with finding the minimum spanning tree (MST) of a connected and non oriented graph.
Given the sequential version of the Prim's algorithm, the project has faced the problem using two different approaches:

- MPI+OpenMP
- CUDA+OpenMP

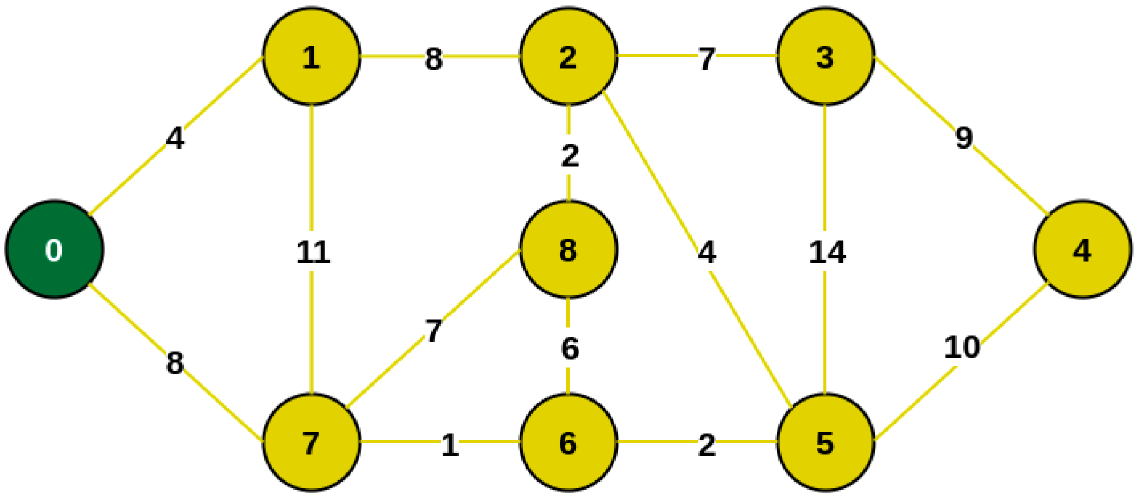
For each approach tests with different problem sizes (graph size) have been carried out in order to analyze the differences between the 2 solutions with respect to the sequential version.

Prim's Algorithm

The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST. (Mehta and Jain) ("Prim's algorithm")
A pseudo-code version is given below:

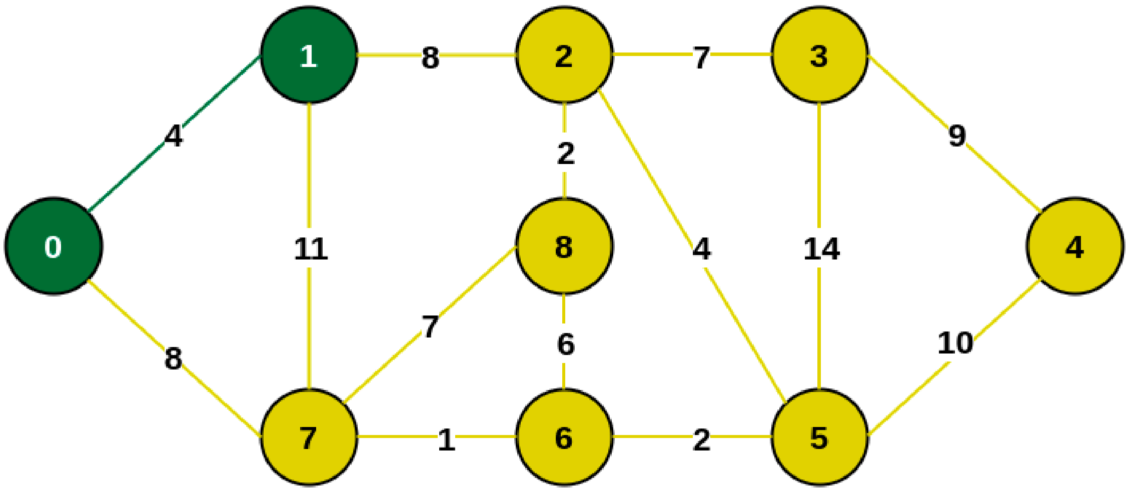
1. Choose an arbitrary vertex as the starting vertex of the MST (ex. vertex 0)
2. Find the edges that connect the current MST with any vertex that is not in the MST
3. Find the minimum edge among these edges
4. Add the edge to the MST if the edge does not create a cycle
5. Repeat from step 2 to step 4 until each vertex is in the MST

Example



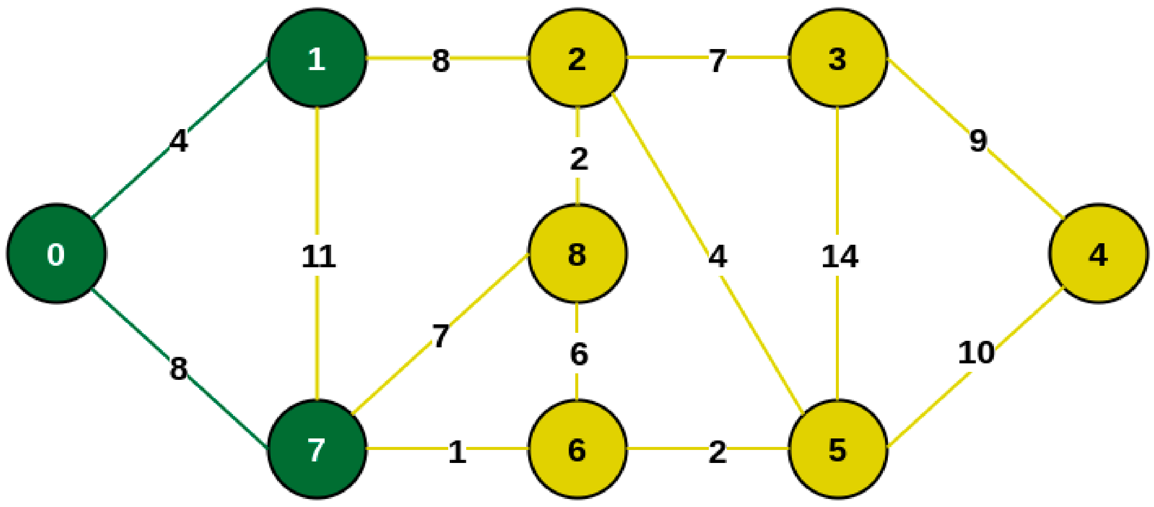
Select an arbitrary starting vertex. Here we have selected 0

Step 1: Firstly, we select an arbitrary vertex that acts as the starting vertex of the Minimum Spanning Tree. Here we have selected vertex 0 as the starting vertex.



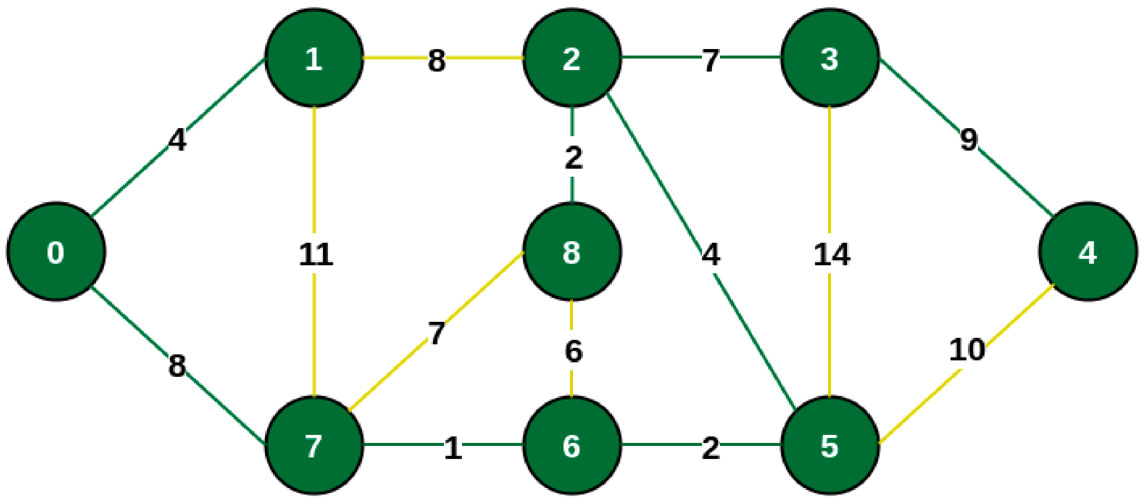
Minimum weighted edge from MST to other vertices is 0-1 with weight 4

Step 2: All the edges connecting the incomplete MST and other vertices are the edges {0, 1} and {0, 7}. Between these two the edge with minimum weight is {0, 1}. So include the edge and vertex 1 in the MST.



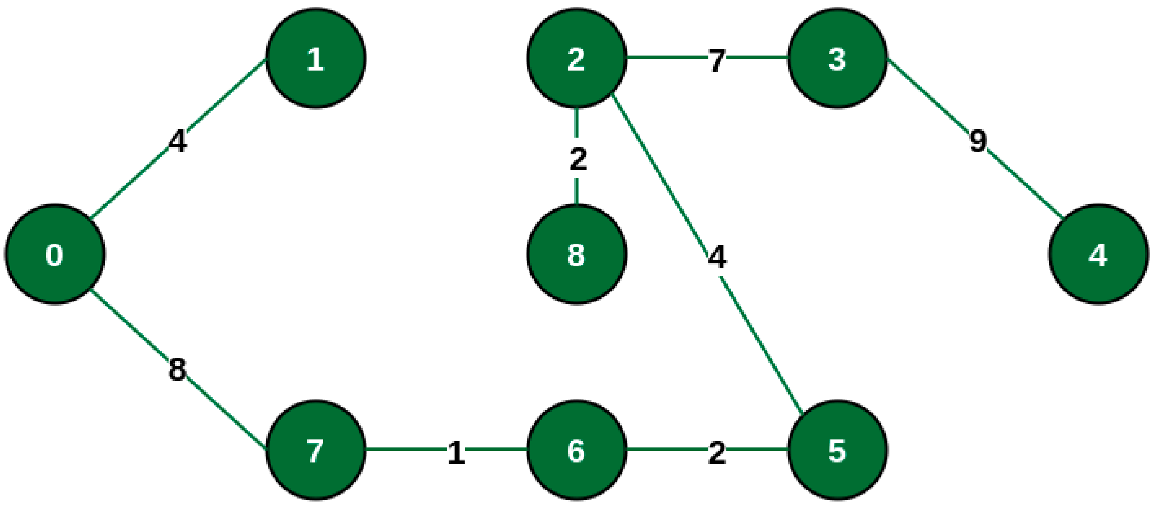
Minimum weighted edge from MST to other vertices is 0-7 with weight 8

Step 3: The edges connecting the incomplete MST to other vertices are {0, 7}, {1, 7} and {1, 2}. Among these edges the minimum weight is 8 which is of the edges {0, 7} and {1, 2}. Let us here include the edge {0, 7} and the vertex 7 in the MST. [We could have also included edge {1, 2} and vertex 2 in the MST].



Minimum weighted edge from MST to other vertices is 3-4 with weight 9

Step X: Only the vertex 4 remains to be included. The minimum weighted edge from the incomplete MST to 4 is {3, 4}.



The final structure of MST

The final structure of the MST is as follows and the weight of the edges of the MST is (4 + 8 + 1 + 2 + 4 + 2 + 7 + 9) = 37.

Experimental Setup

Given the nature of the problem and the hardware resources available, it was necessary to use 2 different hardware, since the hardware used for MPI does not have a dedicated gpu. In particular, the Google Colab environment was used to access an Nvidia GPU and then perform the necessary tests.

Hardware

MPI

For MPI, the following hardware resources were used:

Architecture:	x86_64
CPU op-mode(s):	32-bit, 64-bit
Byte Order:	Little Endian
Address sizes:	39 bits physical, 48 bits virtual
CPU(s):	8
On-line CPU(s) list:	0-7
Thread(s) per core:	2
Core(s) per socket:	4
Socket(s):	1
NUMA node(s):	1
Vendor ID:	GenuineIntel
CPU family:	6
Model:	142
Model name:	Intel(R) Core(TM) i5-8250U CPU @ 1.60GHz
Stepping:	10
CPU MHz:	3071.636
CPU max MHz:	3400,0000
CPU min MHz:	400,0000
BogoMIPS:	3600.00
Virtualization:	VT-x
L1d cache:	128 KiB
L1i cache:	128 KiB
L2 cache:	1 MiB
L3 cache:	6 MiB
NUMA node0 CPU(s):	0-7

The CPU allowed us to use up to 8 different threads.

CUDA

As regards CUDA, the Colab environment has provided a T4 Nvidia GPU with the following specs:

General

Device Type	GPU computing processor - low profile - fanless
Bus Type	PCI Express 3.0 x16
Graphics Engine	NVIDIA Tesla T4
Core Clock	585 MHz
Boost Clock	1590 MHz
CUDA Cores	2560
API Supported	NVIDIA TensorRT, ONNX
Features	Nvidia CUDA technology, Error Correcting Codes (ECC) Memory, UEFI Ready, NVIDIA Turing GPU architecture, 320 NVIDIA Tensor Cores, 8.1 Tflops peak single-precision floating point performance

Memory

Size	16 GB
Technology	GDDR6 SDRAM
Effective Clock Speed	5001 MHz
Bus Width	256-bit
Bandwidth	320 GBps

And as CPU:

```
vendor_id     : GenuineIntel
cpu family    : 6
model         : 63
model name    : Intel(R) Xeon(R) CPU @ 2.30GHz
stepping      : 0
microcode     : 0xffffffff
```

```
cpu MHz         : 2299.998
cache size      : 46080 KB
physical id     : 0
siblings        : 2
core id         : 0
cpu cores       : 1
apicid          : 0
initial apicid  : 0
fpu             : yes
fpu_exception : yes
cpuid level     : 13
wp              : yes
bugs            : cpu_meltdown spectre_v1 spectre_v2 spec_store_bypass l1tf mds swapgs
mmio_stale_data retbleed
bogomips        : 4599.99
clflush size    : 64
cache_alignment         : 64
address sizes  : 46 bits physical, 48 bits virtual
power management:
```

The Colab Environment provided 2 cores of this CPU.

Software

The following software was used for the implementation of parallel algorithms and their analysis:

- Visual Studio Code
- Python 2.7.18
- Ubuntu 20.04.3 LTS
- Gcc 9.4.0
- Google Colab

Measurements

Speedup

In the field of High Performance Computing, the best way to measure the efficiency of an algorithm is the speedup.
In particular, the speedup is calculated as:

$$\frac{T_{seq}}{T_{par}(x)}$$

where x is the number of processes used in the parallel version.

Graphs

During the tests of the algorithm, some graphs have been created in order to test the code with different sizes. In particular, the graph generator allows the creation of graphs with

different vertices. The graph will always be connected (and non directional) and each vertex will have a number of edges that goes from 1, up to half of vertices.
In this sense, the number of edges to analyze will be very large.

Tests

The tests presented in the following paragraphs have been performed in order to give the most accurate performances.
Each version of the algorithm has been runned 10 times for each configuration of the test. This means that the results are the average of multiple iterations.

MPI+OpenMP Solution

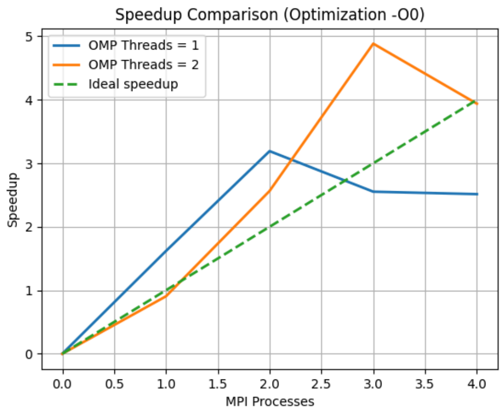
For the development of the parallel algorithm in MPI an analysis of the general functioning of the algorithm was made. The Prim's algorithm is a greedy algorithm that searches the minimum edge among the "available ones" (the edges that connect a vertex in the MST to a vertex that is not in the MST). Moreover, Prim finds the MST for a specific vertex. This means that for each iteration, given the current MST, Prim will find the minimum vertex that connects a vertex to the current MST.
In other words, with a parallel version of the algorithm, it is not possible, at least with this implementation, for the various processes to find their local MST and then merge the partial solutions to create the global one.
The MPI implementation follows these steps:

1. **Read the input** → the Adjacency Matrix is read from the input file and the size of the problem (the rows of the matrix) is given as an argument.
2. **Create the Edge List** → from the matrix, an Edge List will be created. This list contains Edge structures (start,end, weight) and, given that the adjacency list is symmetrical, only the upper triangle of the matrix is used to create the list
3. **Division of the structure** → the Edge List is now splitted in order to be analyzed in parallel among the MPI Processes.
4. **Minimum Local Search** → each process execute a function in order to find their minimum edge
5. **Global Edge** → the algorithm will now find global minimum edge by executing a custom operation
6. **Update the MST** → the MST is updated by the Process with rank 0 with a Broadcast operation.
7. **Iterate**→ the steps for 4 to 6 will be made until each vertex is added to the MST

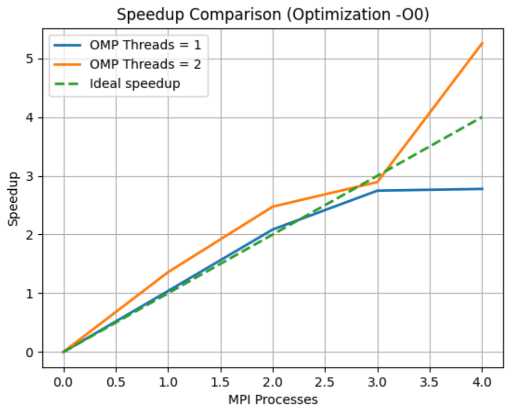
On the other hand, OpenMP deals with the parallelization of the Minimum Local Search, in order to reduce the amount of time needed to execute that portion of the code.
Given the provided solution it is clear that the main issue with the algorithm is the amount of communications, between the processes, needed to update the current MST.

Results

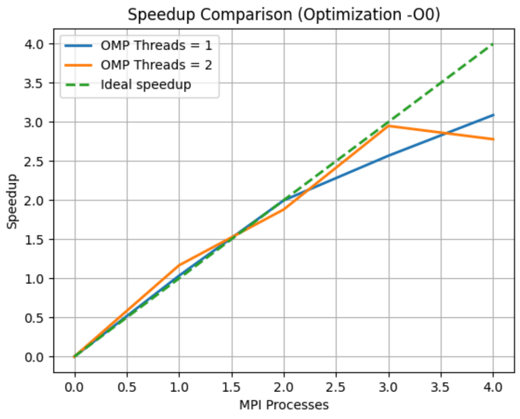
Case OPT_0



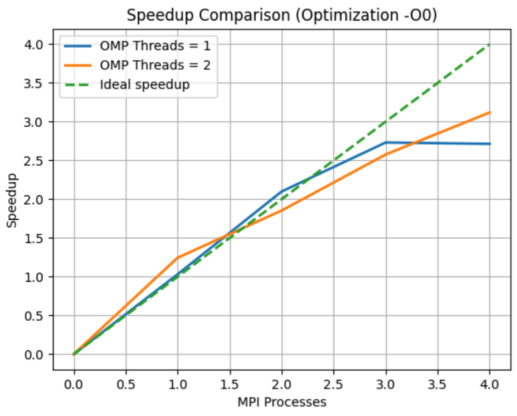
(500)



(750)

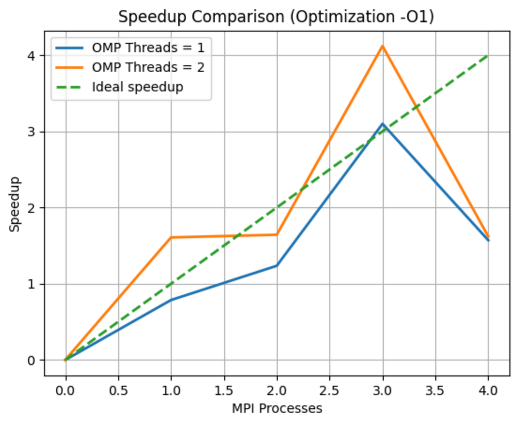


(1000)

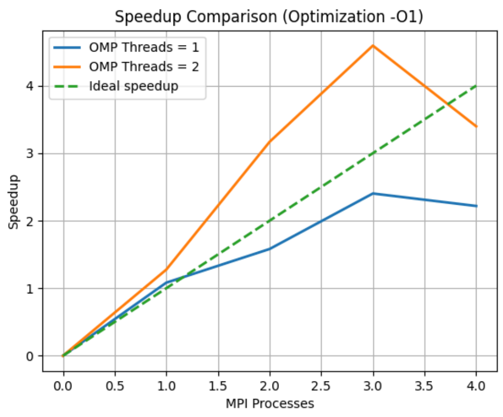


(2000)

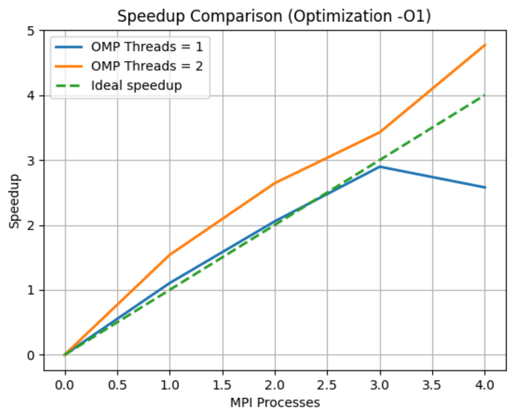
Case OPT_1



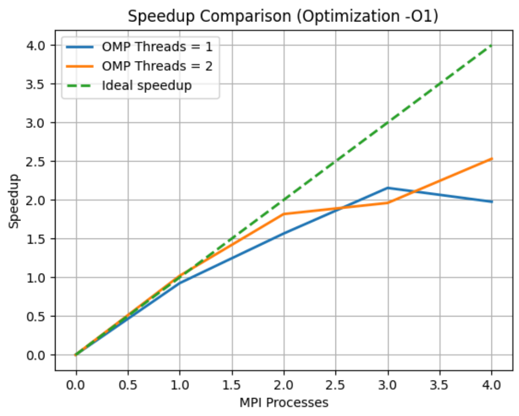
(500)



(750)

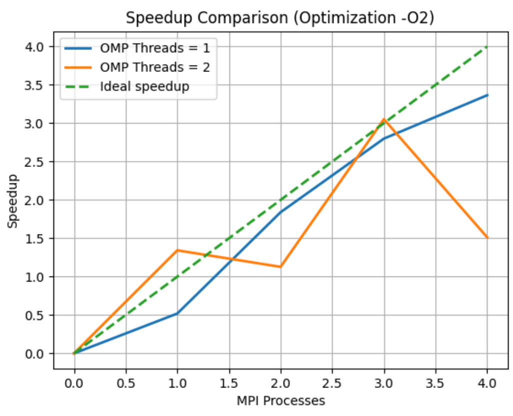


(1000)

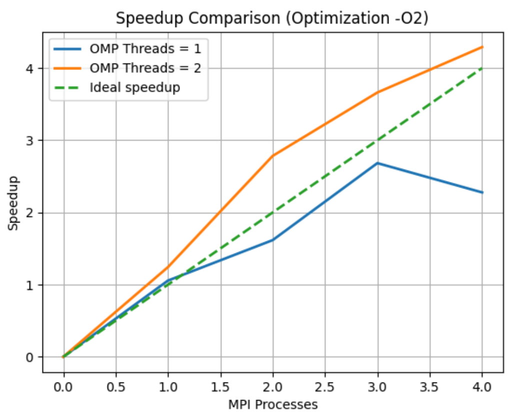


(2000)

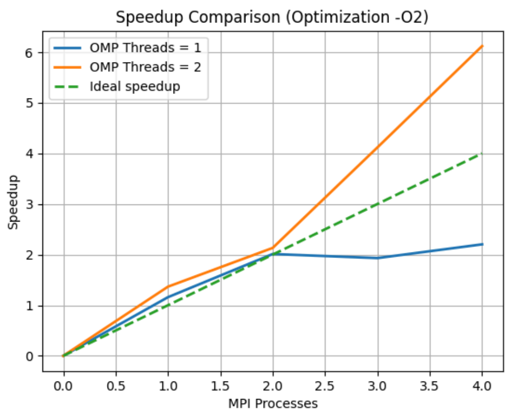
Case OPT_2



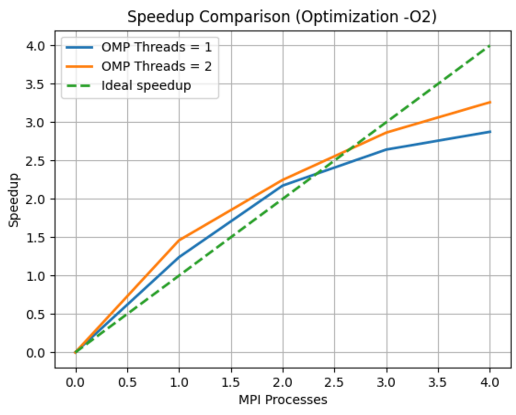
(500)



(750)

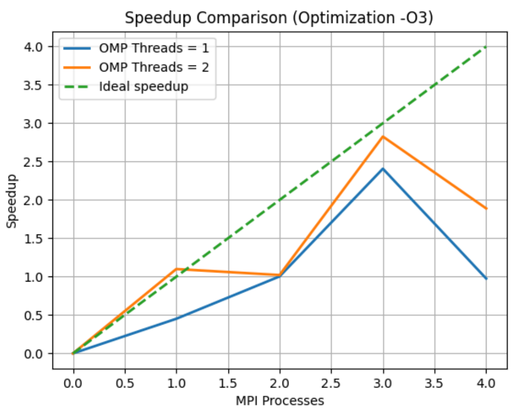


(1000)

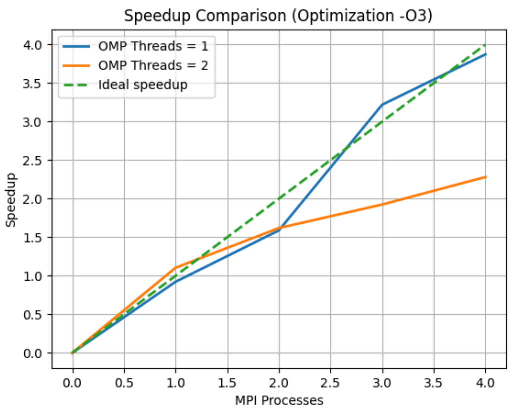


(2000)

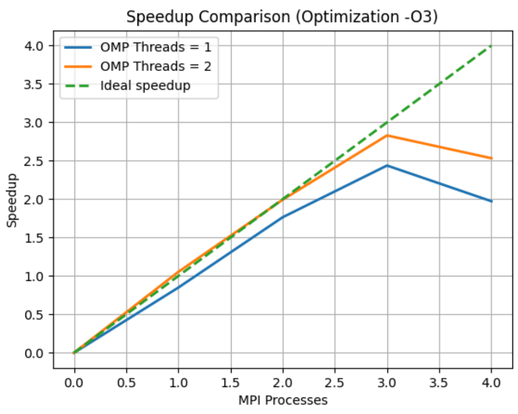
Case OPT_3



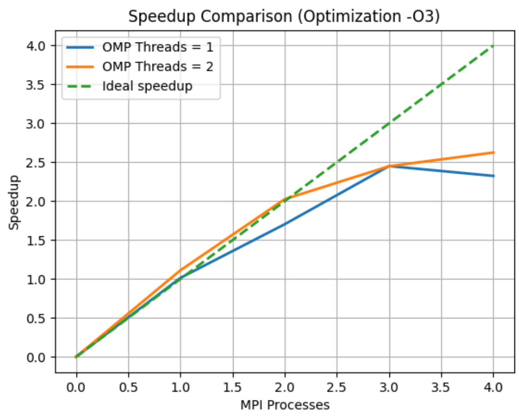
(500)



(750)



(1000)



(2000)

Analysis

By looking at the graphs above it is possible to make some considerations about the results. In particular, you can see that, especially for the test without O0, there are some results that exceed the ideal speedup.
This could have happened because of a particular execution (untruthful results) or maybe because the OpenMP parts give an additional sprint to the performances and add an additional speedup.
The first test (OPT_O0) shows also that as we increase the size of the problem, the performances start to follow the ideal speedup.
The other tests show that with the optimization options, the parallel version will essentially follow the ideal speedup, at least for a size that is not too big (up to 1000 vertices).
When the size of the problem grows, we can see, especially for the O3 optimization that the performances get worse.
We can also see that in some cases the obtained speedup, in particular when we use 2 OMP_THREADS, changes a lot from 2 MPI processes to 3 MPI Processes.
In particular this happens when the problem size is 500 and we can see that the algorithm performs better when 1 MPI process, than the speedup decreases with 2 MPI processes and then increases with 3.
Another consideration can be made with respect to the best configuration that can be used.

In many tests it is clearly visible that when we perform the algorithm with more than 3 MPI processes, the speedup slowly starts to get worse.
Unluckily, the experimental setup has allowed to perform the tests only with 4 MPI processes and only 2 OMP_THREADS.
With a more powerful cpu we could set up more tests with more MPI processes and see if the problem of the communications among them is something that could make performances worse.
In conclusion, we can say that the hybrid implementation with MPI+OpenMP has obtained good results, except for the anomalous ones that exceed the ideal speedup.

CUDA+OpenMP Solution

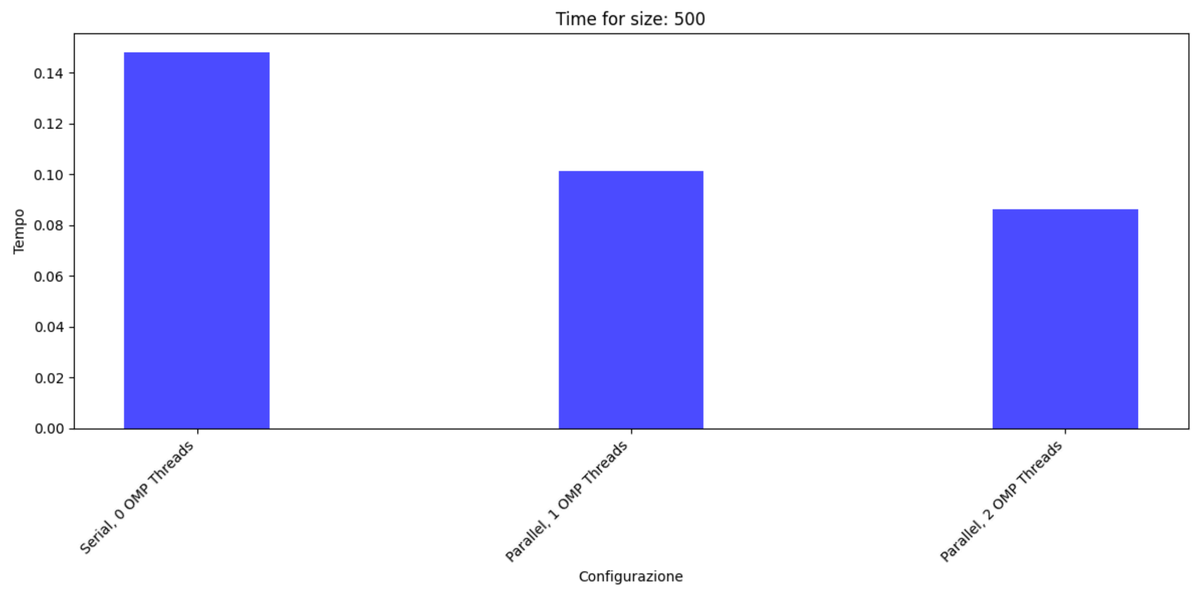
For the CUDA+OpenMP implementation, a similar resources management has been made. The main differences concern the use of OpenMP and the management of shared data.
Starting with OpenMP, the structure of the algorithm allowed to parallelize only some parts. In particular, OpenMP has been used to speed up the malloc operations from the host to the device. The remaining part of the code couldn't be parallelized with OpenMP given that it is executed by the GPU's threads.
The main structure of the graph, in particular the Edge structure and the edge, is the same of the MPI version.
Regarding the CUDA part, the algorithm follows some steps:

1. **Dividing the data among the threads** → each thread works on a small portion of the list
2. **Find the local minimum edge** → each thread find its local minimum edge for the current MST and, if there is an available edge, it will write it in a structure
3. **Find the global minimum edge** → given that more blocks could work in parallel, it is needed to find the global minimum edge among the blocks. In this sense, a shared structure is used to store the local min edge of each block and then a global min will be found among the edges in the structure
4. **Update** → the global minimum edge will be added to the MST with a communication between the device and the host. Then the host update the MST and send the current MST back to the device
5. **Iterate** → the kernel will be executed until the MST contains every vertices.

Another aspect faced was the decision of how many blocks to use. In particular before launching the kernel an operation is made in order to use a number of blocks in function of the problem size. The number of blocks is calculated as: (rows -1 + BLOCK_SIZE -1) / BLOCK_SIZE; so it is function of the vertices and the block size.

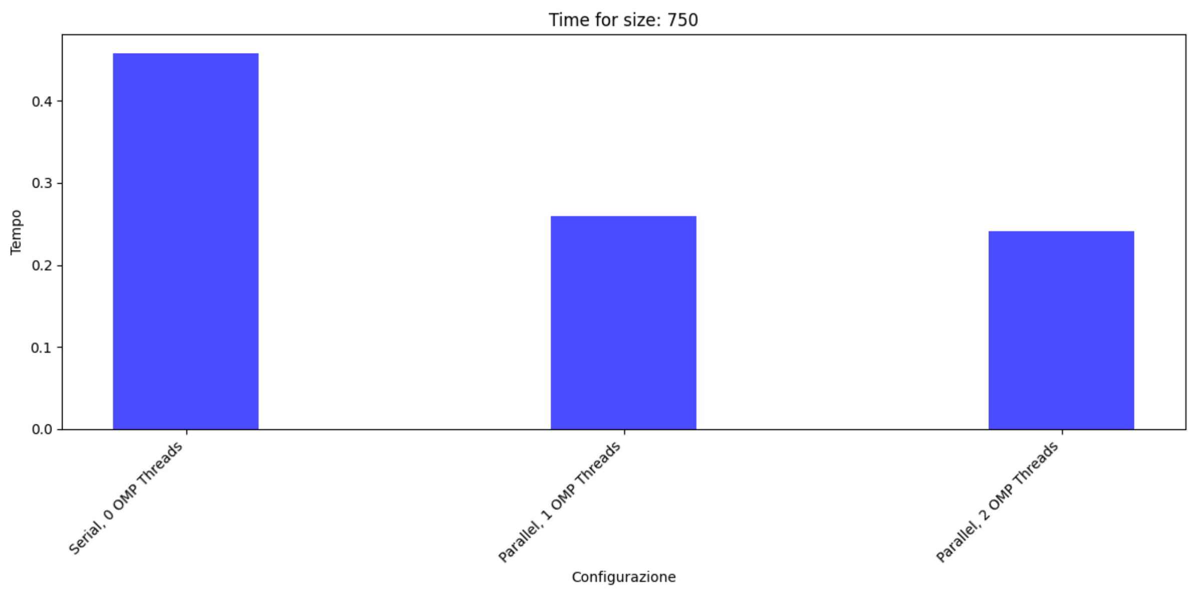
Results

Size 500



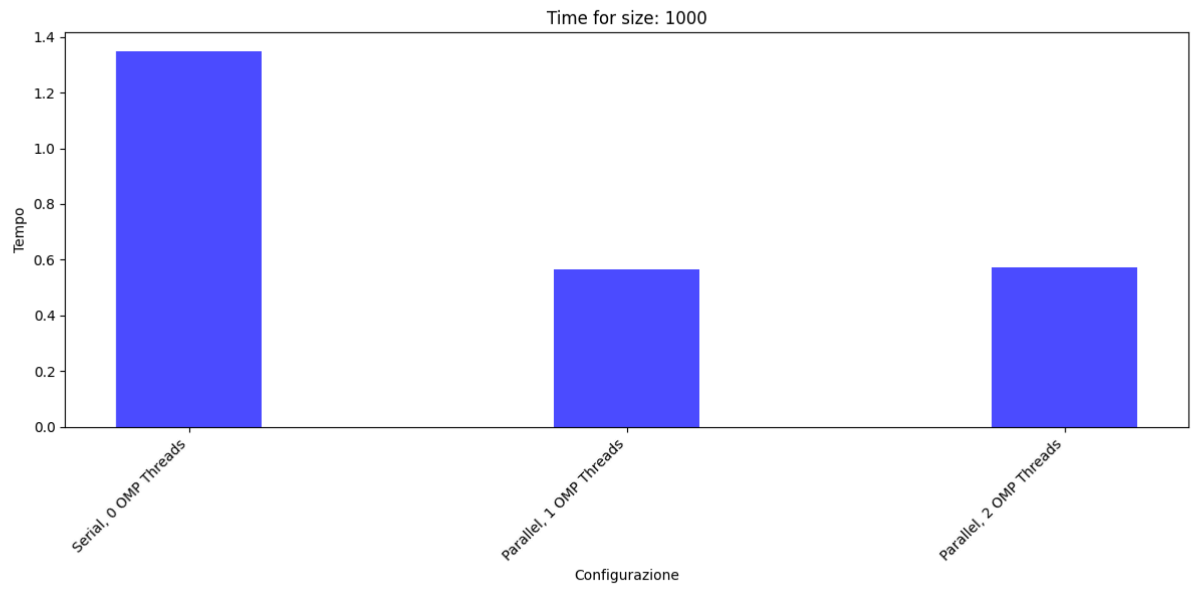
Version	OpenMP Threads	Time	Speedup
Serial	0	0.147935	1.0
Parallel	1	0.101128	1,462849063
Parallel	2	0.086074	1,7187

Size 750



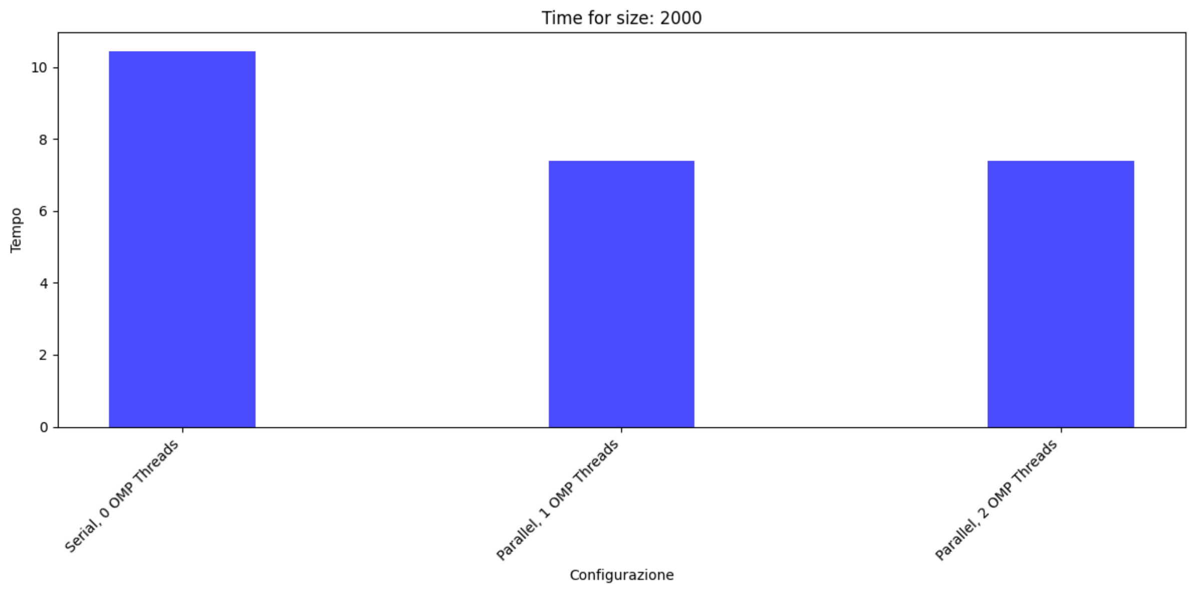
Version	OpenMP Threads	Time	Speedup
Serial	0	0.458159	1.0
Parallel	1	0.25897	1,76915859
Parallel	2	0.241672	1,895788507

Size 1000



Version	OpenMP Threads	Time	Speedup
Serial	0	1.348408	1.0
Parallel	1	0.566440	2,380495728
Parallel	2	0.574156	2,348504588

Size 2000



Version	OpenMP Threads	Time	Speedup
Serial	0	10.433293	1.0
Parallel	1	7.388172	1,412161628
Parallel	2	7.392812	1,411275304

Analysis

By looking at graphs and tables we can say that also for the CUDA+OpenMP version of the parallel algorithm we have some improvements in terms of execution time.
In particular, also in this case we can see that the best performances are obtained for a problem size that is not too big. This tells that the amount of time needed to copy the data from the host to the device is a bottleneck with respect to the rest of the algorithm, even if it is parallelized with OpenMP.
We can see that, for size equal to 1000, the parallel version takes almost a third of the time with respect to the sequential version.
In general, as we increase the problem size, the parallel version is faster than the sequential one; at least for these tests. But, when the size of the problem grows, also the parallel version starts to get worse.
Another visible thing is that using 1 or 2 OMP_THREADS does not increase the performance much.

Performances Comparison

By looking at the results, we can say that the parallel version of the Prim's Algorithm offers some benefits in terms of speedup.
Unfortunately, the experimental setup did not allow us to evaluate the actual differences of MPI and CUDA, given that two different hardware have been used.
This was forced because the Colab Environment provides a CPU with only 2 cores; this means that the performances of the MPI version would not have been so good.
On the other hand, the speedup has been measured with respect to a sequential version runned on the same hardware.
So we can still make some considerations about the results and the differences.
In general, we can say that the MPI version performed better with respect to the CUDA one and that the hybrid with OpenMP does not increase the performance of the CUDA version much because the main part is executed by the GPU.
Further analysis would be a good thing to make some tests according to the block size, even though the decision of using a block_size of 256 was made after some preliminary tests that showed that after 256 the performances were the same and below were worse.

Bibliography

Mehta, Divyanshu, and Sandeep Jain. “Prim’s Algorithm for Minimum Spanning Tree (MST).” *GeeksforGeeks*, 4 December 2023, https://www.geeksforgeeks.org/prims-minimum-spanning-tree-mst-greedy-algo-5/. Accessed 12 January 2024.

“Prim's algorithm.” *Wikipedia*, https://en.wikipedia.org/wiki/Prim%27s_algorithm. Accessed 13 January 2024.