

**Report Common Assignment: Parallel Tarjan**

**Lecturer: Francesco Moscato -** [**fmoscato@unisa.it**](http://fmoscato@unisa.it)

* **Marcone Giuseppe - 0622701896 –** [**g.marcone2@studenti.unisa.it**](mailto:g.marcone2@studenti.unisa.it)
* **Pizzulo Rocco Gerardo - 0622701990 –** [**r.pizzulo@studenti.unisa.it**](mailto:r.pizzulo@studenti.unisa.it)
* **Russo Luigi - 0622702071 –** [**l.russo86@studenti.unisa.it**](mailto:l.russo86@studenti.unisa.it)



# Index

[Index 2](#_Toc1)

[Problem description 3](#_Toc2)

[Solution 3](#_Toc3)

[Experimental setup 4](#_Toc4)

[Hardware 4](#_Toc5)

[Memory Device 5](#_Toc6)

[Software 6](#_Toc7)

[Performance, SpeedUp & Efficiency 7](#_Toc8)

[Case study 1: No OpenMP 7](#_Toc9)

[6000 vertices and 150 edges 8](#_Toc10)

[6000 vertices and 250 edges 9](#_Toc11)

[6000 vertices and 400 edges 10](#_Toc12)

[Case study 2: send without pack 11](#_Toc13)

[6000 vertices and 150 edges 12](#_Toc14)

[6000 vertices and 250 edges 13](#_Toc15)

[6000 vertices and 400 edges 14](#_Toc16)

[Case study 3: send with pack 15](#_Toc17)

[6000 vertices and 150 edges 16](#_Toc18)

[6000 vertices and 250 edges 17](#_Toc19)

[6000 vertices and 400 edges 18](#_Toc20)

[Final consideration 19](#_Toc21)

# Problem description

The requirement was to provide a parallell version of the Tarjan's algorithm to find Strongly Connected Components in a Graph. The implementation is an hibrid  of message passing / shared memory paradigm implemented by using MPI and openMP.

There are no constraints on the graph structure and allocation type.

# Solution

In the proposed solution the graph is read from a text file and splitted into subgraphs each of which is sent to an MPI process.

After that a reduction mechanism was implemented; one half of the processes elaborates the partial graph and forwards the result to the others, the mechanism continues until the whole graph converges to process 0. The library used for the comunication between processes is MPI (Message Passing Interface).

In our solution there are two versions provided for the parallel program with 2 different communication systems: in the first case, communication was via the functions MPI\_Send() and MPI\_Recv () by sending the fields separately, in the second case, data were sent after encapsulation via MPI\_Pack().

In addition, a second level of parallelization was added via OpenMP trying to make the loops more efficient.

Initially, measurements were made with MPI and later with the addition of OpenMP.

# Experimental setup

## Hardware

*Architecture: x86\_64*

*CPU op-mode(s): 32-bit, 64-bit*

*Address sizes: 39 bits physical, 48 bits virtual*

*Byte Order: Little Endian*

*CPU(s): 8*

*On-line CPU(s) list: 0-7*

*Vendor ID: GenuineIntel*

*Model name: Intel(R) Core(TM) i7-10510U CPU @ 1.80GHz*

*CPU family: 6*

*Model: 142*

*Thread(s) per core: 2*

*Core(s) per socket: 4*

*Socket(s): 1*

*Stepping: 12*

*CPU(s) scaling MHz: 39%*

*CPU max MHz: 4900.0000*

*CPU min MHz: 400.0000*

*BogoMIPS: 4599.93*

*Flags: fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe syscall nx pdpe1gb rdtscp lm constant\_tsc art arch\_perfmon pebs bts rep\_good nopl xtopology n*

*onstop\_tsc cpuid aperfmperf pni pclmulqdq dtes64 monitor ds\_cpl vmx est tm2 ssse3 sdbg fma cx16 xtpr pdcm pcid sse4\_1 sse4\_2 x2apic movbe popcnt tsc\_deadline\_timer aes xsave avx f16c rdrand lahf\_lm abm 3dnowprefet*

*ch cpuid\_fault epb invpcid\_single ssbd ibrs ibpb stibp ibrs\_enhanced tpr\_shadow vnmi flexpriority ept vpid ept\_ad fsgsbase tsc\_adjust sgx bmi1 avx2 smep bmi2 erms invpcid mpx rdseed adx smap clflushopt intel\_pt xs*

*aveopt xsavec xgetbv1 xsaves dtherm ida arat pln pts hwp hwp\_notify hwp\_act\_window hwp\_epp md\_clear flush\_l1d arch\_capabilities*

## Memory Device

*Array Handle: 0x1000*

*Error Information Handle: Not Provided*

*Total Width: 64 bits*

*Data Width: 64 bits*

*Size: 8 GB*

*Form Factor: Row Of Chips*

*Set: None*

*Locator: MotherBoard*

*Bank Locator: BANK 0*

*Type: LPDDR3*

*Type Detail: Synchronous*

*Speed: 2133 MT/s*

*Manufacturer: Samsung*

*Serial Number: 55000000*

*Asset Tag: 01000000*

*Part Number: K4EBE304EC-EGCG*

*Rank: 2*

*Configured Memory Speed: 2133 MT/s*

*Minimum Voltage: Unknown*

*Maximum Voltage: Unknown*

*Configured Voltage: 1.2 V*

*Memory Technology: DRAM*

*Memory Operating Mode Capability: Volatile memory*

*Firmware Version: 55000000*

*Module Manufacturer ID: Bank 1, Hex 0xCE*

*Module Product ID: Unknown*

*Memory Subsystem Controller Manufacturer ID: Unknown*

*Memory Subsystem Controller Product ID: Unknown*

*Non-Volatile Size: None*

*Volatile Size: 8 GB*

*Cache Size: None*

*Logical Size: None*

## Software

*OS: Linux fedora 6.1.6-200.fc37.x86\_64*

*GCC: gcc (GCC) 12.2.1 20221121 (Red Hat 12.2.1-4)*

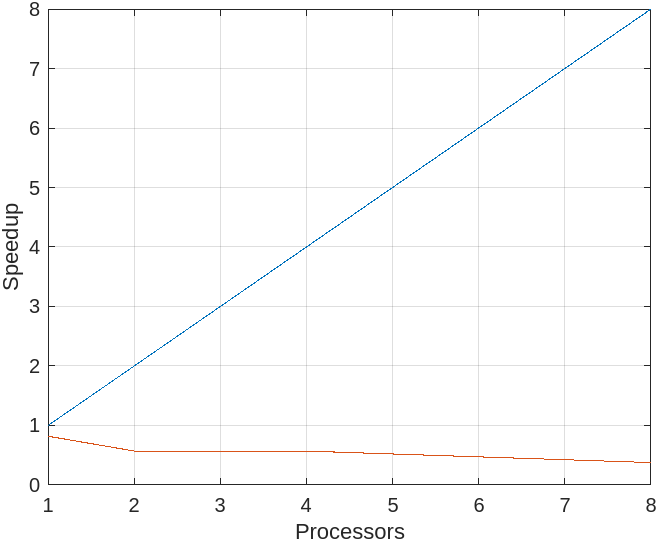
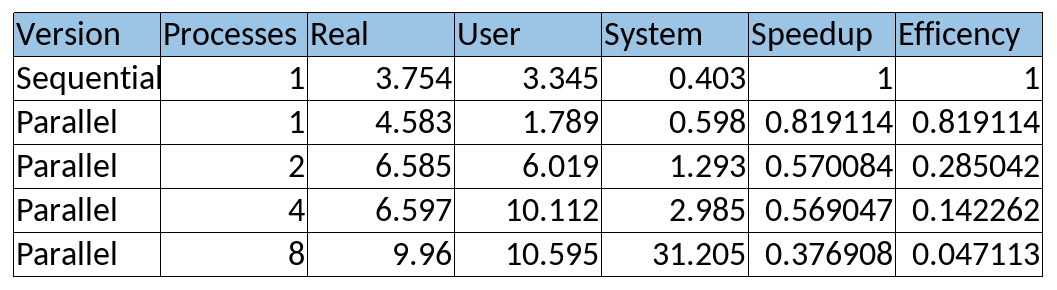
*Swap: 8GB*

# Performance, SpeedUp & Efficiency

## Case study 1: No OpenMP

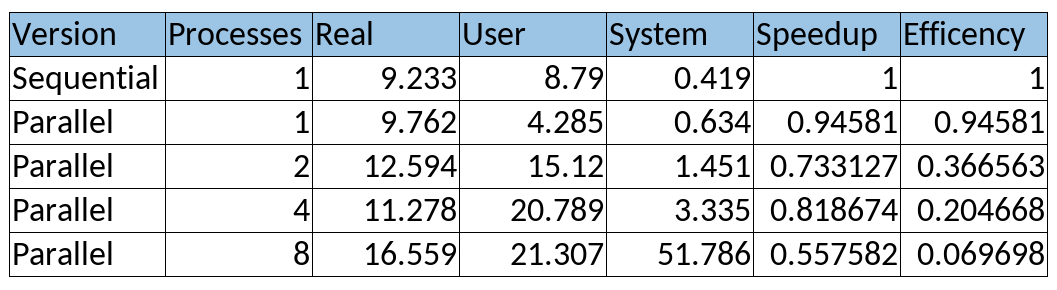
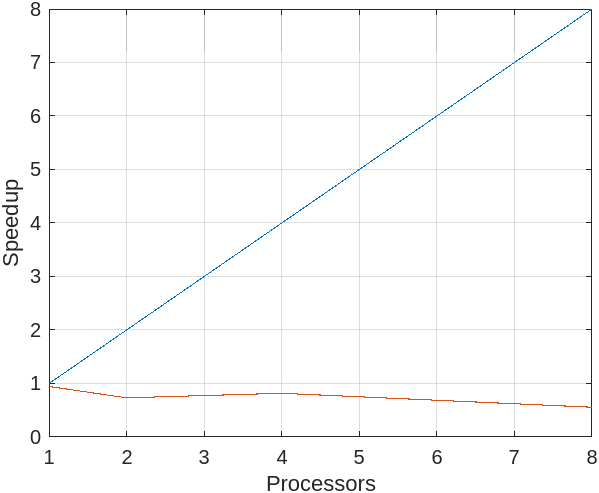
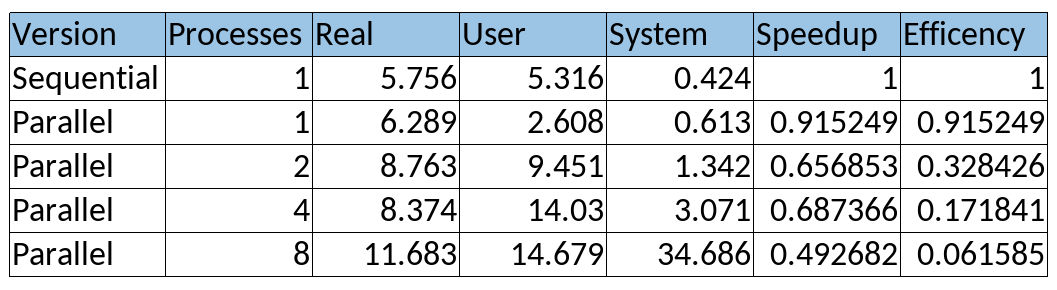
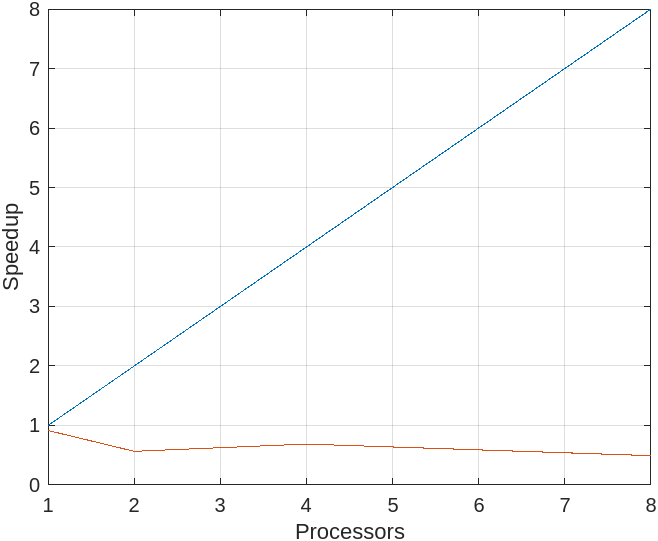
...

### 6000 vertices and 150 edges



### 6000 vertices and 250 edges

### 6000 vertices and 400 edges



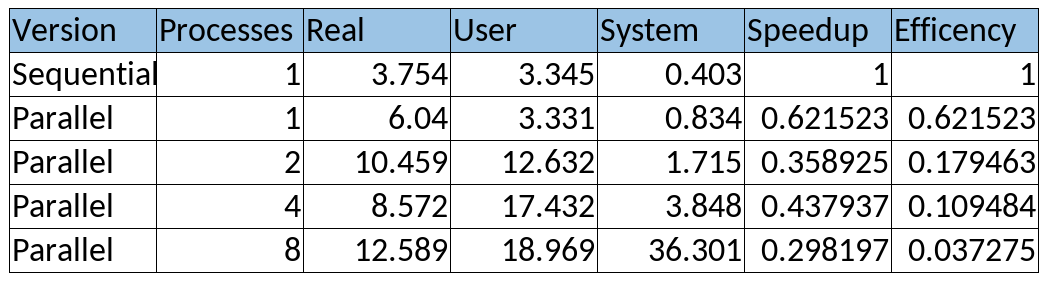
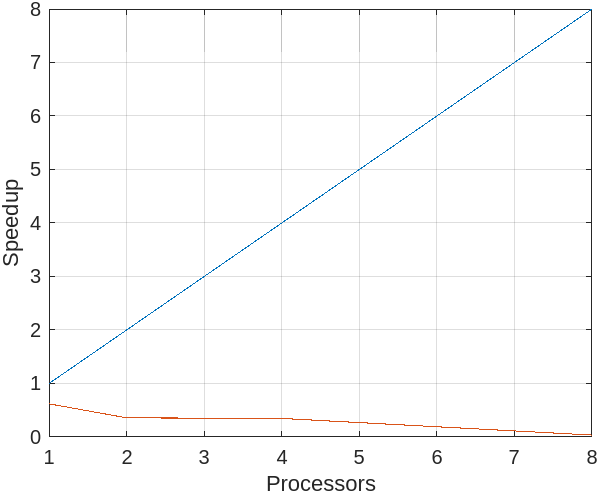
## Case study 2: send without pack

In the first case study, communication was implemented by means of MPI\_Send() and MPI\_Recv() without encapsulating the data, thus sending them separately.

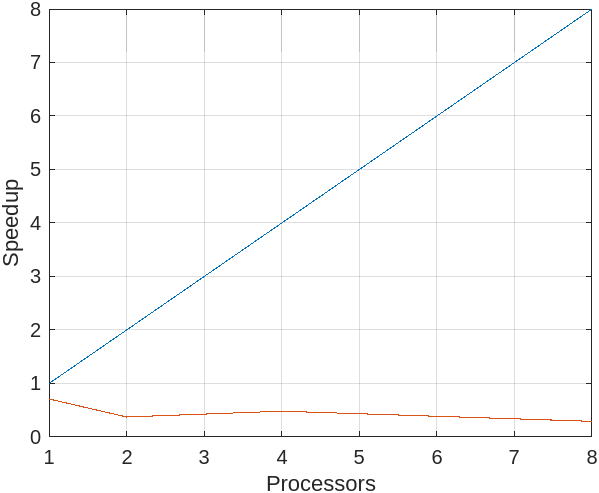
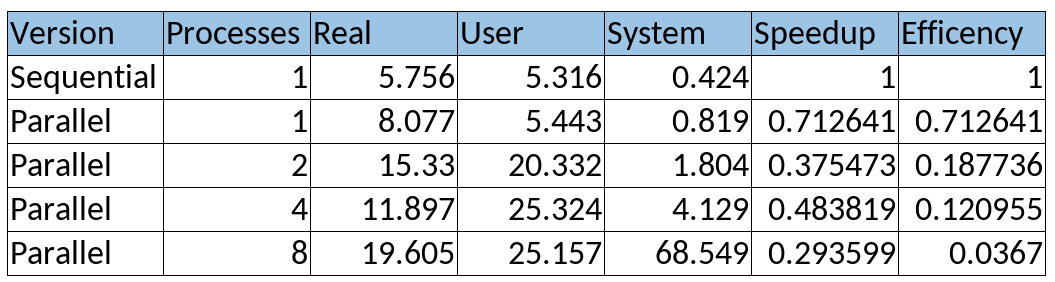
Both sequential and parallel compilation were done with the -O3 gcc optimization and measurements were performed with 1, 2, 4 and 8 MPI processes and 4 OpenMP threads.

The graph used has 6000 vertices and 150 arcs for the first measurement, 250 for the second and 400 for the third.

### 6000 vertices and 150 edges

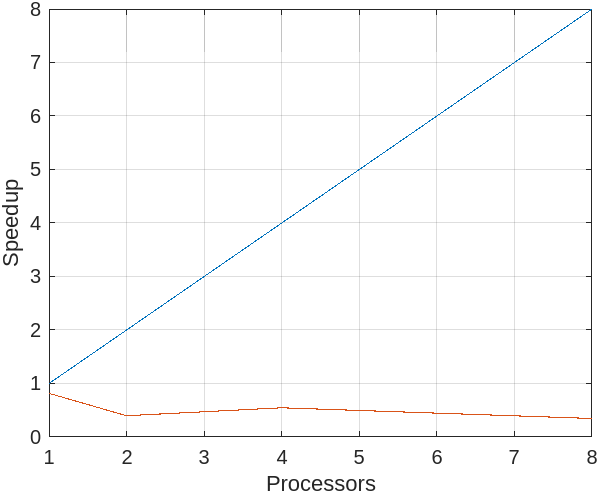
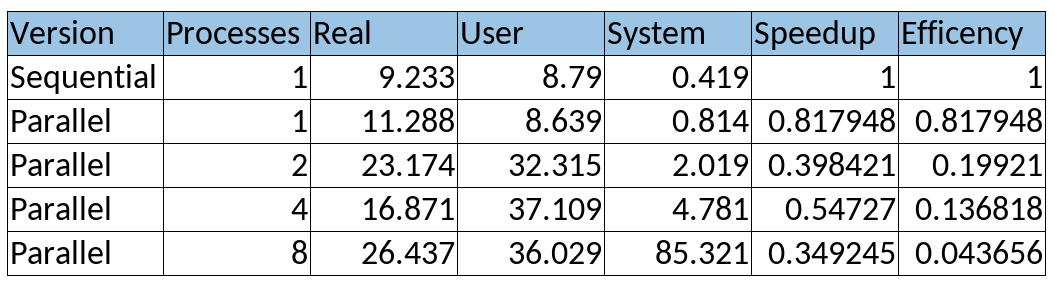


### 6000 vertices and 250 edges



### 6000 vertices and 400 edges

## Case study 3: send with pack

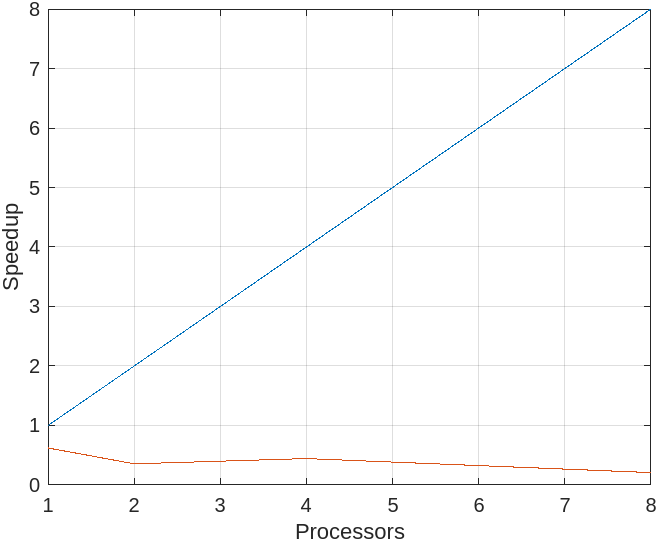
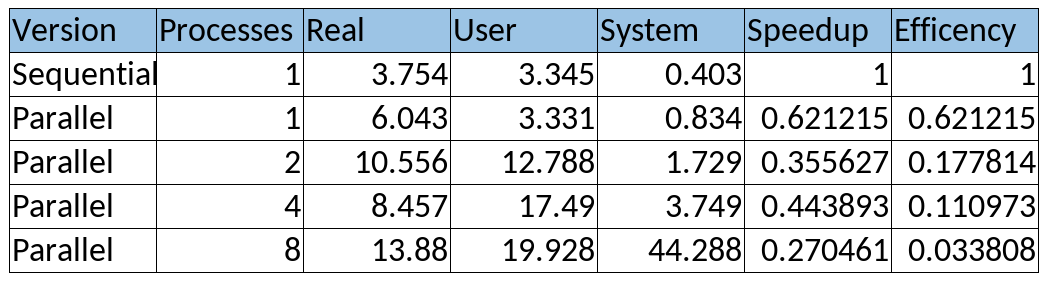


In the second case study, the submitted vertex fields were first encapsulated with MPI\_Pack() and then submitted with MPI\_Send(), so the number of submissions was reduced by 1/3.

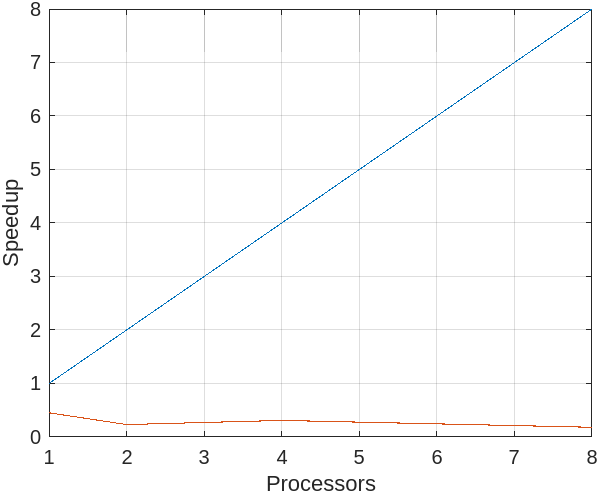
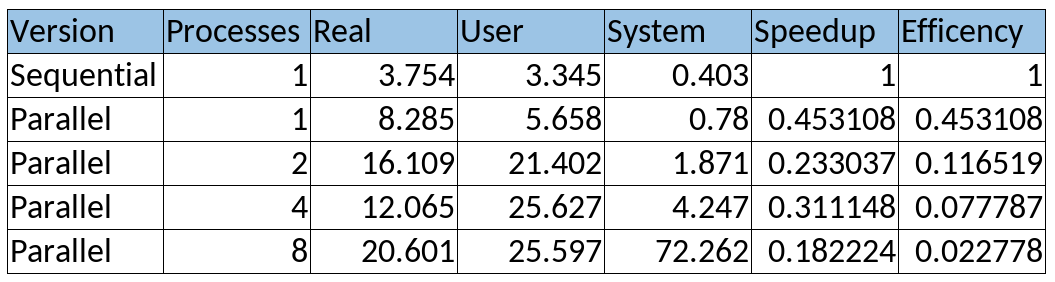
Again, this was compiled with the gcc -O3 option, measurements were performed with 1, 2, 4 and 8 MPI processes and 4 OpenMP threads.

The datasets used to measure the performances of this parallel version are the same of the first case.

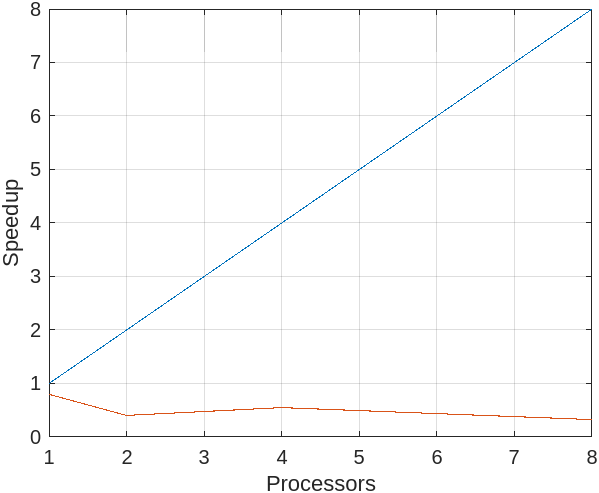
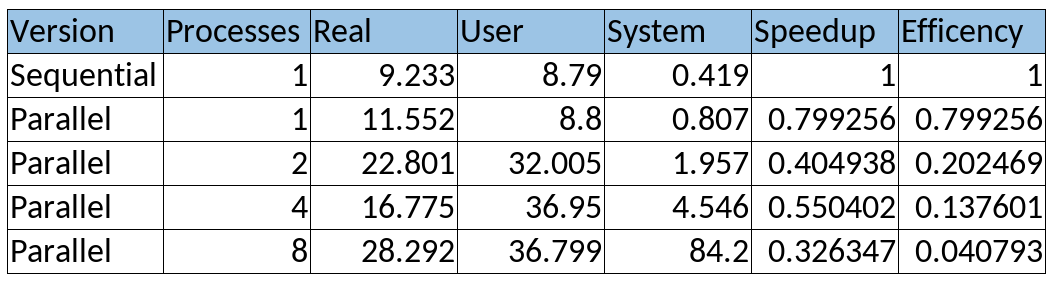
### 6000 vertices and 150 edges



### 6000 vertices and 250 edges



### 6000 vertices and 400 edges



# Final consideration

As expected, we can say that the Tarjan’s Algorithm runs much better without parrallelization.  
In fact, in the light of the data collected, we can see that the speedup of the parallel version is in each case much less than 1. This is due to the multiple execution of Tarjan on several graphs which, although smaller than the original graph, must still be reprocessed to combine the SCCs found on them.