

## Report Common Assignment: Parallel Tarjan

Lecturer: Francesco Moscato - [fmoscato@unisa.it](mailto: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
Problem description .....	3
Solution .....	3
Experimental setup .....	4
Hardware .....	4
Memory Device .....	5
Software .....	6
Performance, SpeedUp & Efficiency .....	7
Case study 1: No OpenMP .....	7
6000 vertices and 150 edges .....	8
6000 vertices and 250 edges .....	9
6000 vertices and 400 edges .....	10
Case study 2: send without pack .....	11
6000 vertices and 150 edges .....	12
6000 vertices and 250 edges .....	13
6000 vertices and 400 edges .....	14
Case study 3: send with pack .....	15
6000 vertices and 150 edges .....	16
6000 vertices and 250 edges .....	17
6000 vertices and 400 edges .....	18
Final consideration .....	19

## Problem description

The requirement was to provide a parallel version of the Tarjan's algorithm to find Strongly Connected Components in a Graph. The implementation is an hybrid 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 communication 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

aveo opt 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

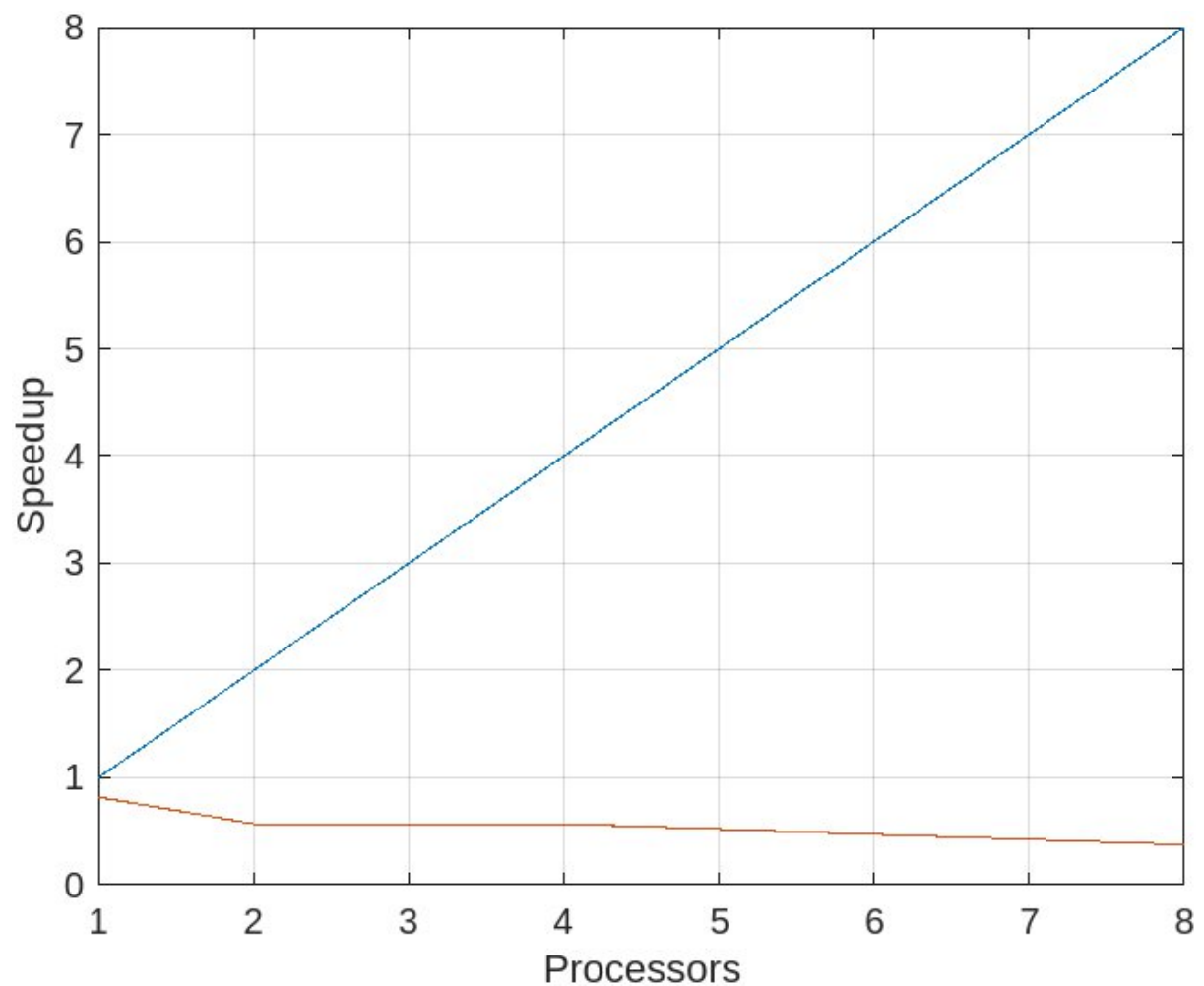
In the first case study, communication was implemented using `MPI_Send()` and `MPI_Recv()` without encapsulating the data, thus sending them separately. In this version, our solution has been implemented without using openMP.

Both sequential and parallel compilation were done with the `-O3` gcc optimization and measurements were performed with 1, 2, 4 and 8 MPI processes. Note that the source code is the same, but it has been compiled without `“-fopenmp”` in order to ignore the `#pragma` directives.

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

6000 vertices and 150 edges

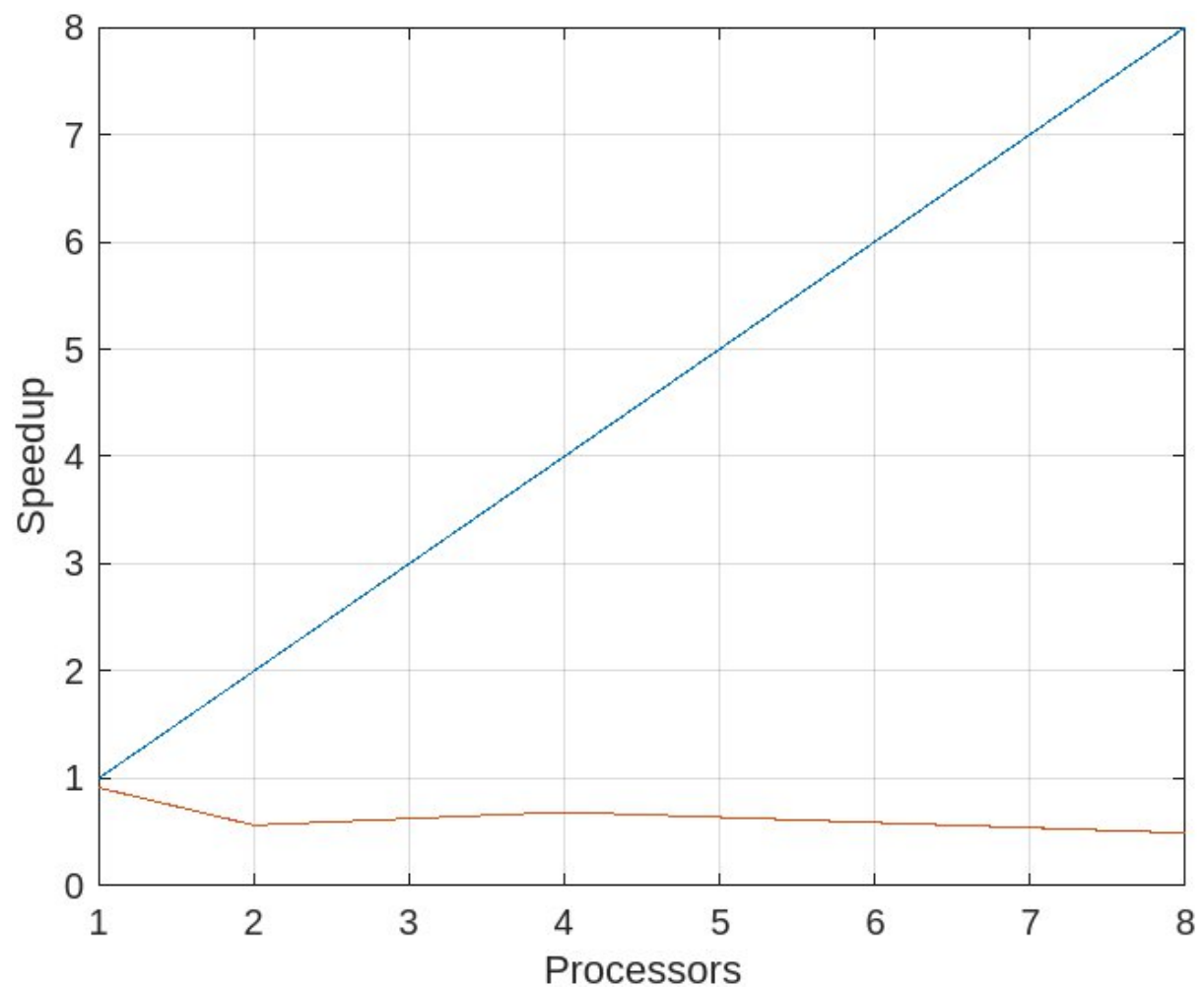
Version	Processes	Real	User	System	Speedup	Efficiency
Sequential	1	3.754	3.345	0.403	1	1
Parallel	1	4.583	1.789	0.598	0.819114	0.819114
Parallel	2	6.585	6.019	1.293	0.570084	0.285042
Parallel	4	6.597	10.112	2.985	0.569047	0.142262
Parallel	8	9.96	10.595	31.205	0.376908	0.047113





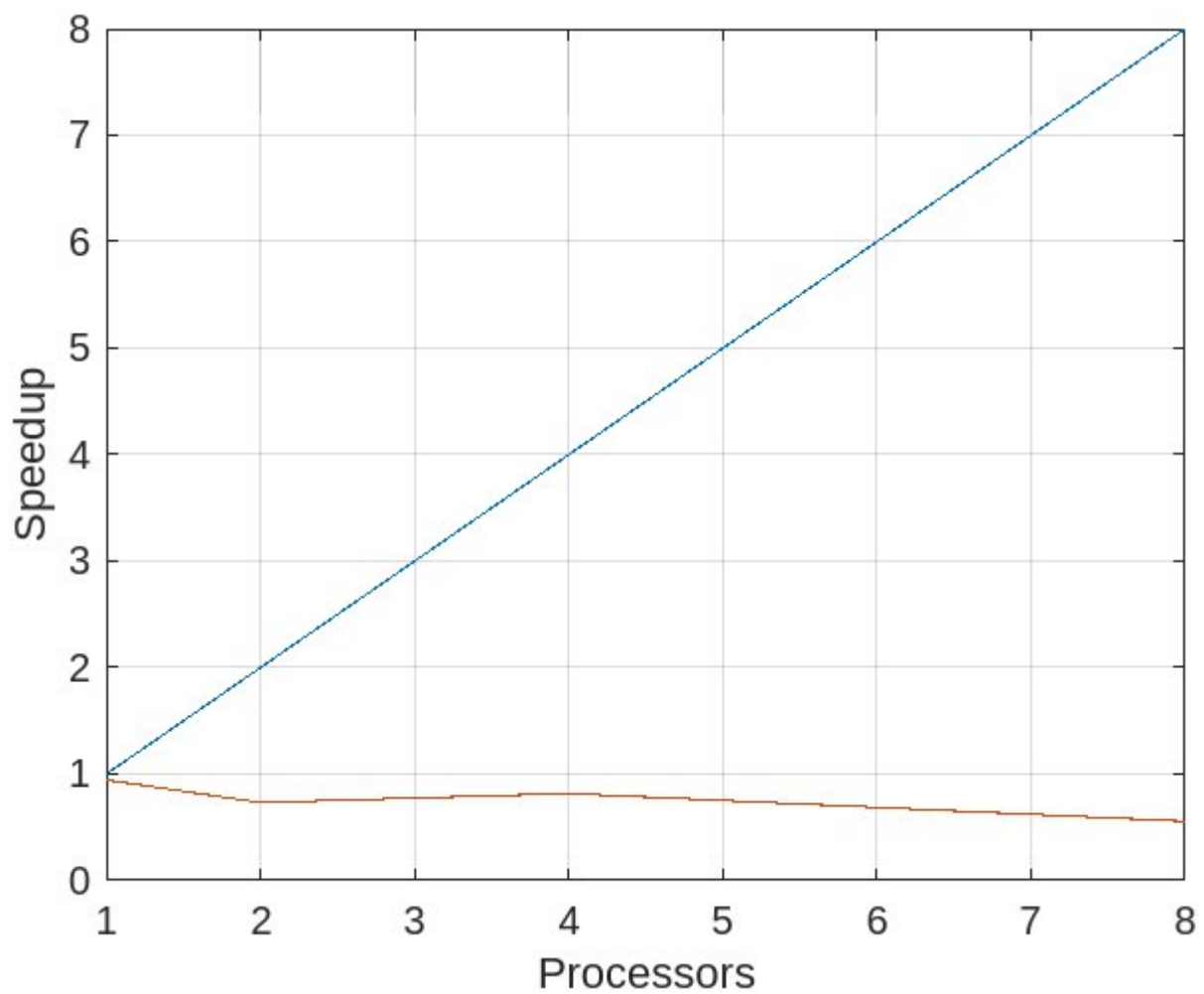
6000 vertices and 250 edges

Version	Processes	Real	User	System	Speedup	Efficiency
Sequential	1	5.756	5.316	0.424	1	1
Parallel	1	6.289	2.608	0.613	0.915249	0.915249
Parallel	2	8.763	9.451	1.342	0.656853	0.328426
Parallel	4	8.374	14.03	3.071	0.687366	0.171841
Parallel	8	11.683	14.679	34.686	0.492682	0.061585



6000 vertices and 400 edges

Version	Processes	Real	User	System	Speedup	Efficency
Sequential	1	9.233	8.79	0.419	1	1
Parallel	1	9.762	4.285	0.634	0.94581	0.94581
Parallel	2	12.594	15.12	1.451	0.733127	0.366563
Parallel	4	11.278	20.789	3.335	0.818674	0.204668
Parallel	8	16.559	21.307	51.786	0.557582	0.069698



## Case study 2: send without pack

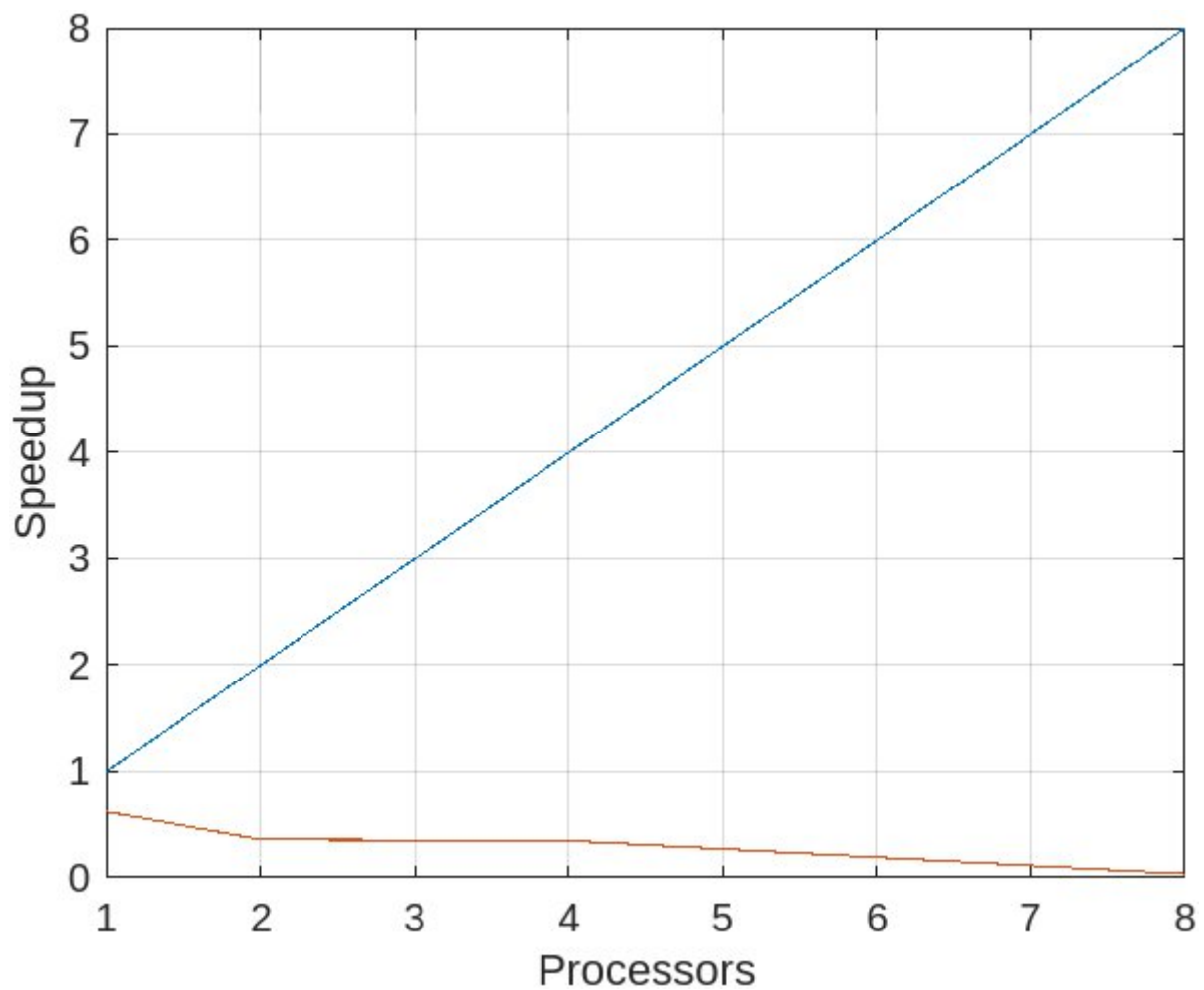
In the second case study, communication was implemented by means of `MPI_Send()` and `MPI_Recv()` without encapsulating the data, thus sending them separately.

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 edges for the first measurement, 250 for the second and 400 for the third.

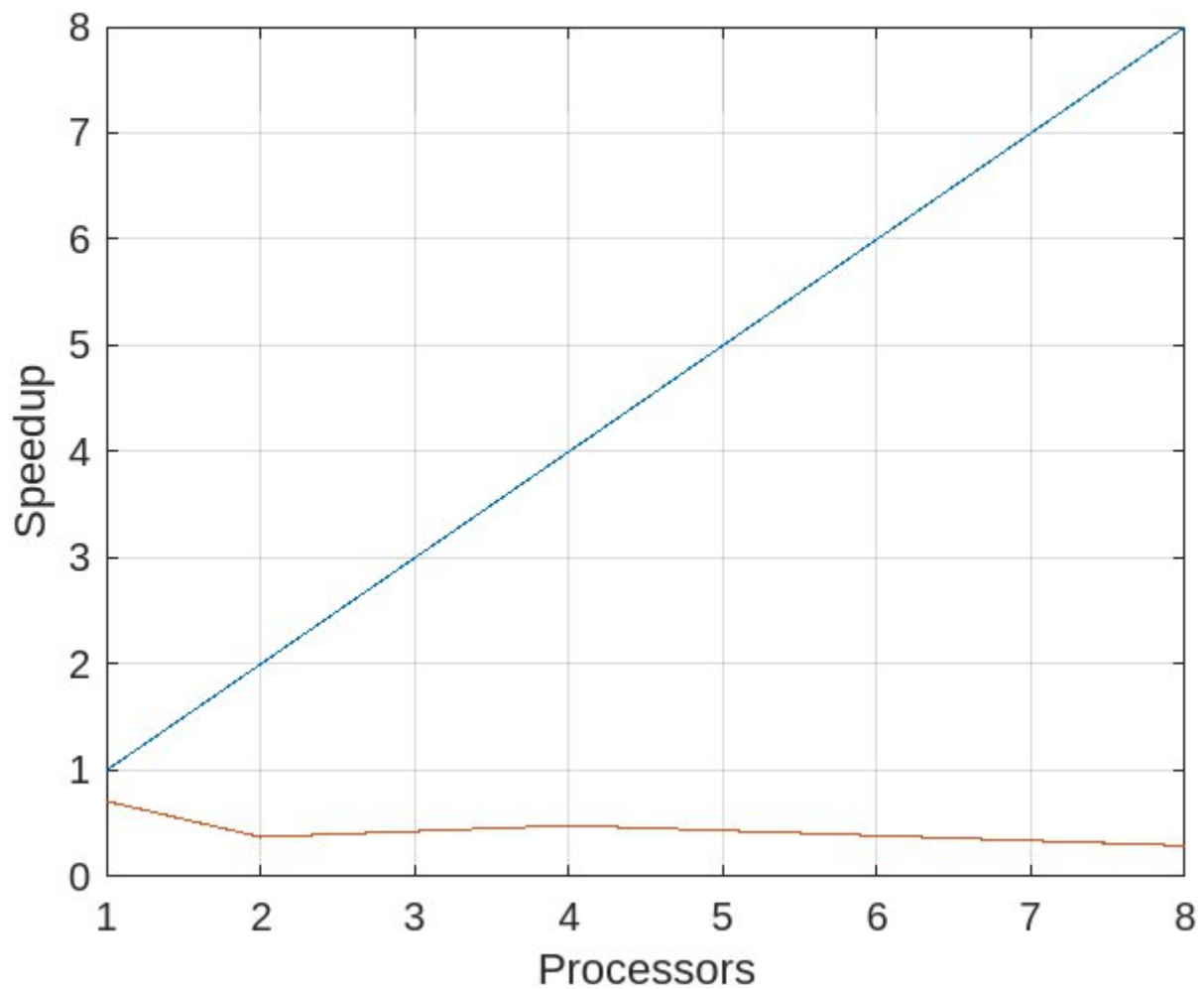
6000 vertices and 150 edges

Version	Processes	Real	User	System	Speedup	Efficency
Sequential	1	3.754	3.345	0.403	1	1
Parallel	1	6.04	3.331	0.834	0.621523	0.621523
Parallel	2	10.459	12.632	1.715	0.358925	0.179463
Parallel	4	8.572	17.432	3.848	0.437937	0.109484
Parallel	8	12.589	18.969	36.301	0.298197	0.037275



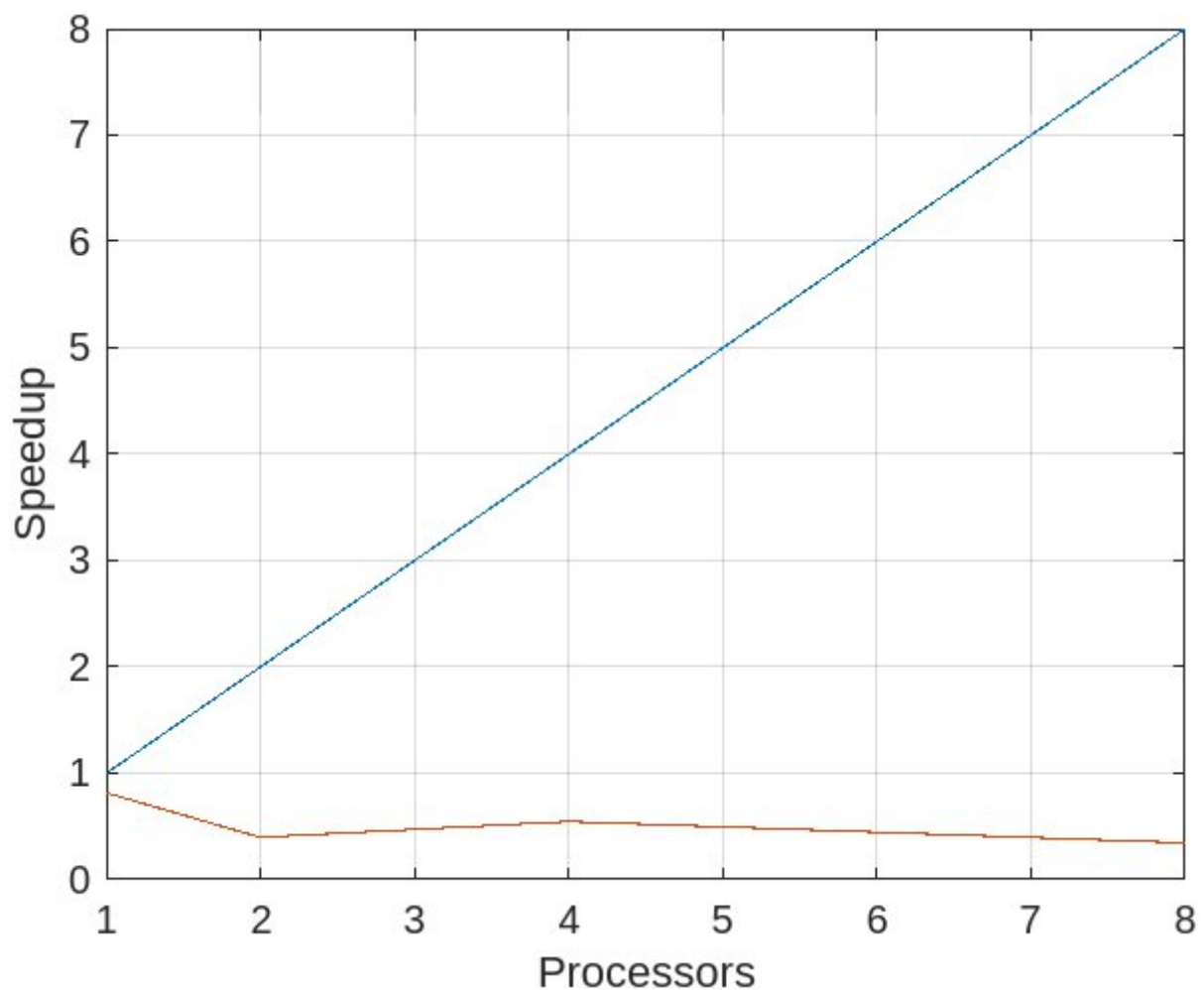
6000 vertices and 250 edges

Version	Processes	Real	User	System	Speedup	Efficiency
Sequential	1	5.756	5.316	0.424	1	1
Parallel	1	8.077	5.443	0.819	0.712641	0.712641
Parallel	2	15.33	20.332	1.804	0.375473	0.187736
Parallel	4	11.897	25.324	4.129	0.483819	0.120955
Parallel	8	19.605	25.157	68.549	0.293599	0.0367



6000 vertices and 400 edges

Version	Processes	Real	User	System	Speedup	Efficiency
Sequential	1	9.233	8.79	0.419	1	1
Parallel	1	11.288	8.639	0.814	0.817948	0.817948
Parallel	2	23.174	32.315	2.019	0.398421	0.19921
Parallel	4	16.871	37.109	4.781	0.54727	0.136818
Parallel	8	26.437	36.029	85.321	0.349245	0.043656



## Case study 3: send with pack

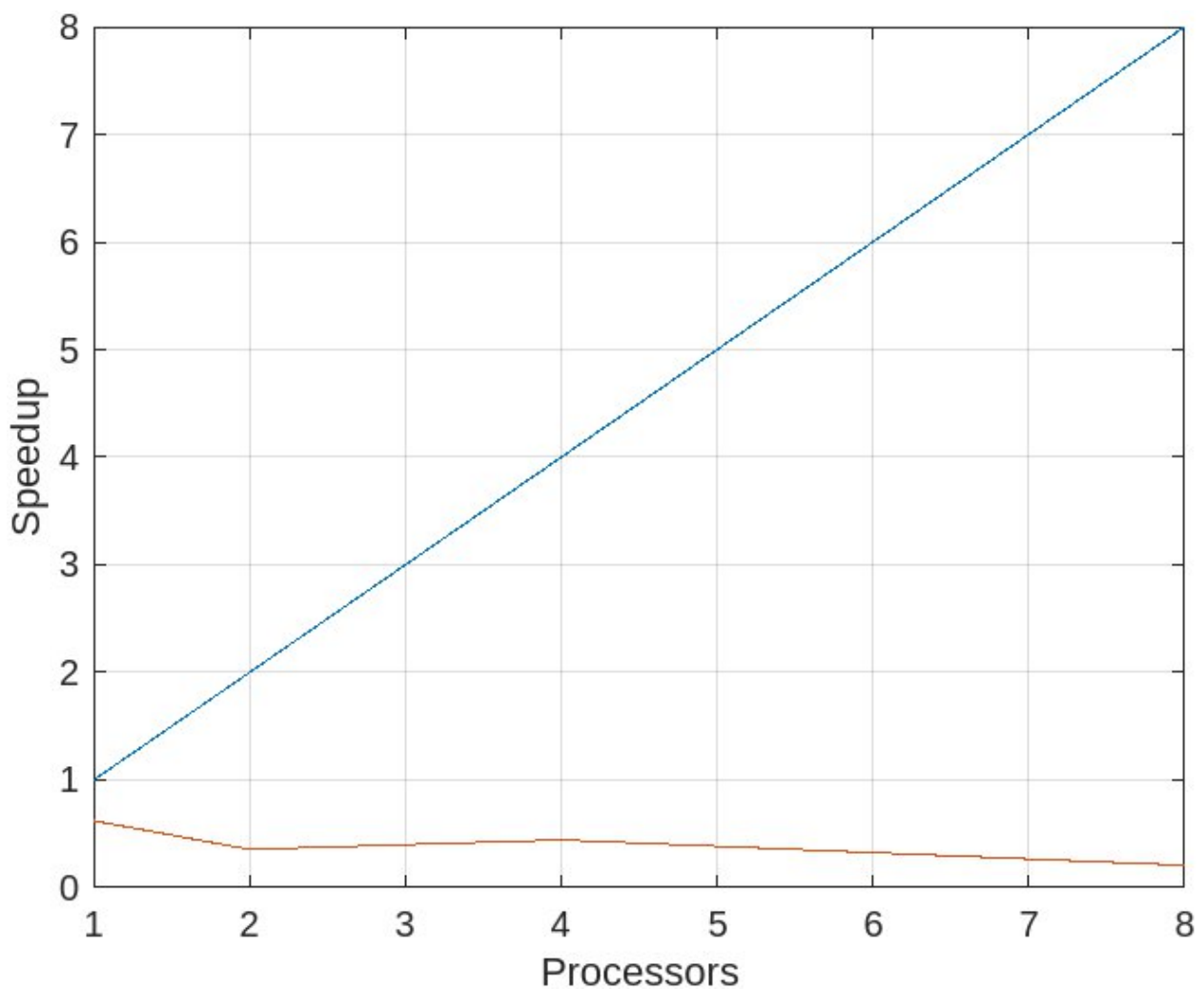
In the last case study, the submitted vertex fields were first encapsulated with `MPI_Pack()` and then submitted with `MPI_Send()`, so the number of submissions was reduced by 1/3.

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 two cases.

6000 vertices and 150 edges

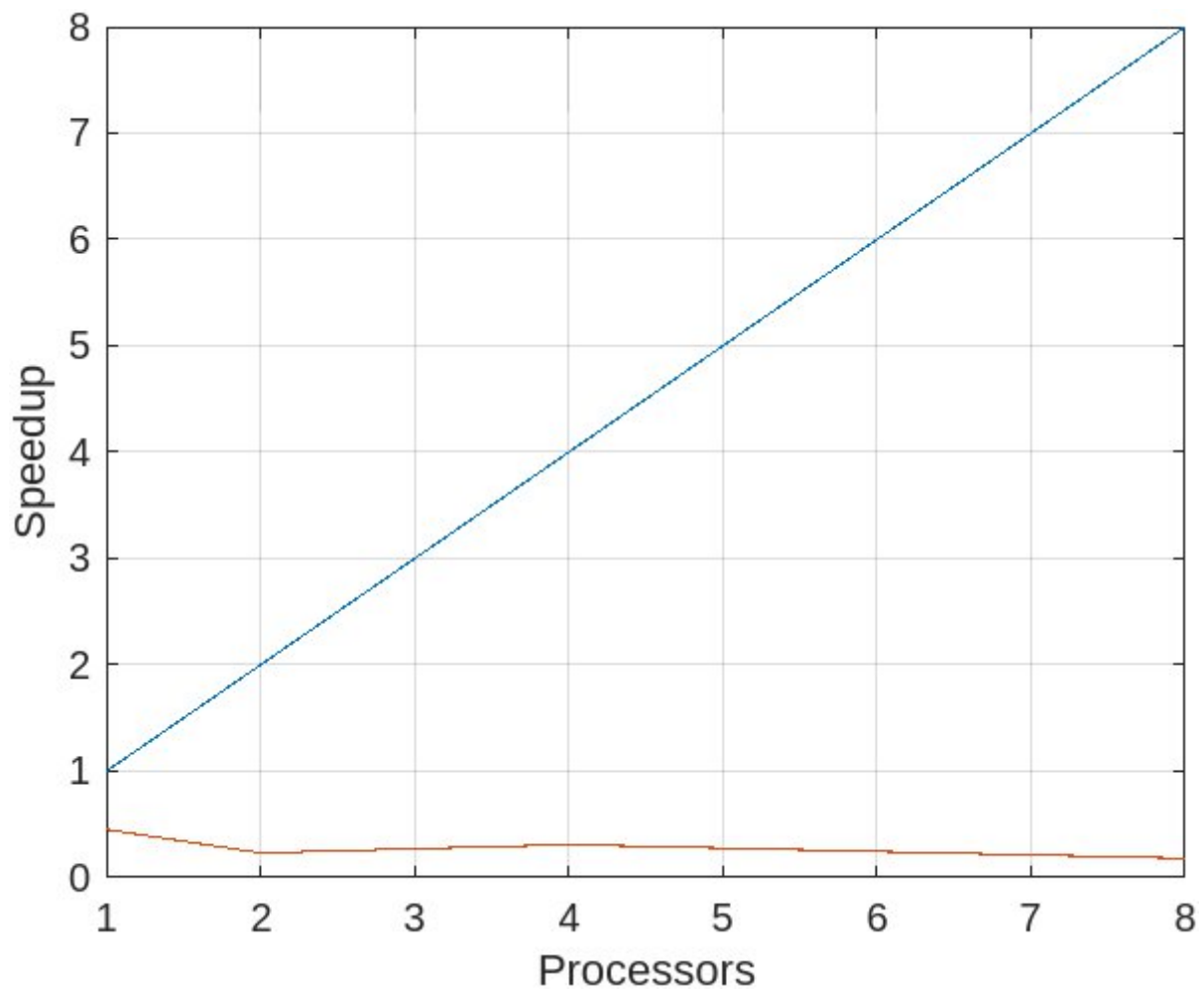
Version	Processes	Real	User	System	Speedup	Efficiency
Sequential	1	3.754	3.345	0.403	1	1
Parallel	1	6.043	3.331	0.834	0.621215	0.621215
Parallel	2	10.556	12.788	1.729	0.355627	0.177814
Parallel	4	8.457	17.49	3.749	0.443893	0.110973
Parallel	8	13.88	19.928	44.288	0.270461	0.033808





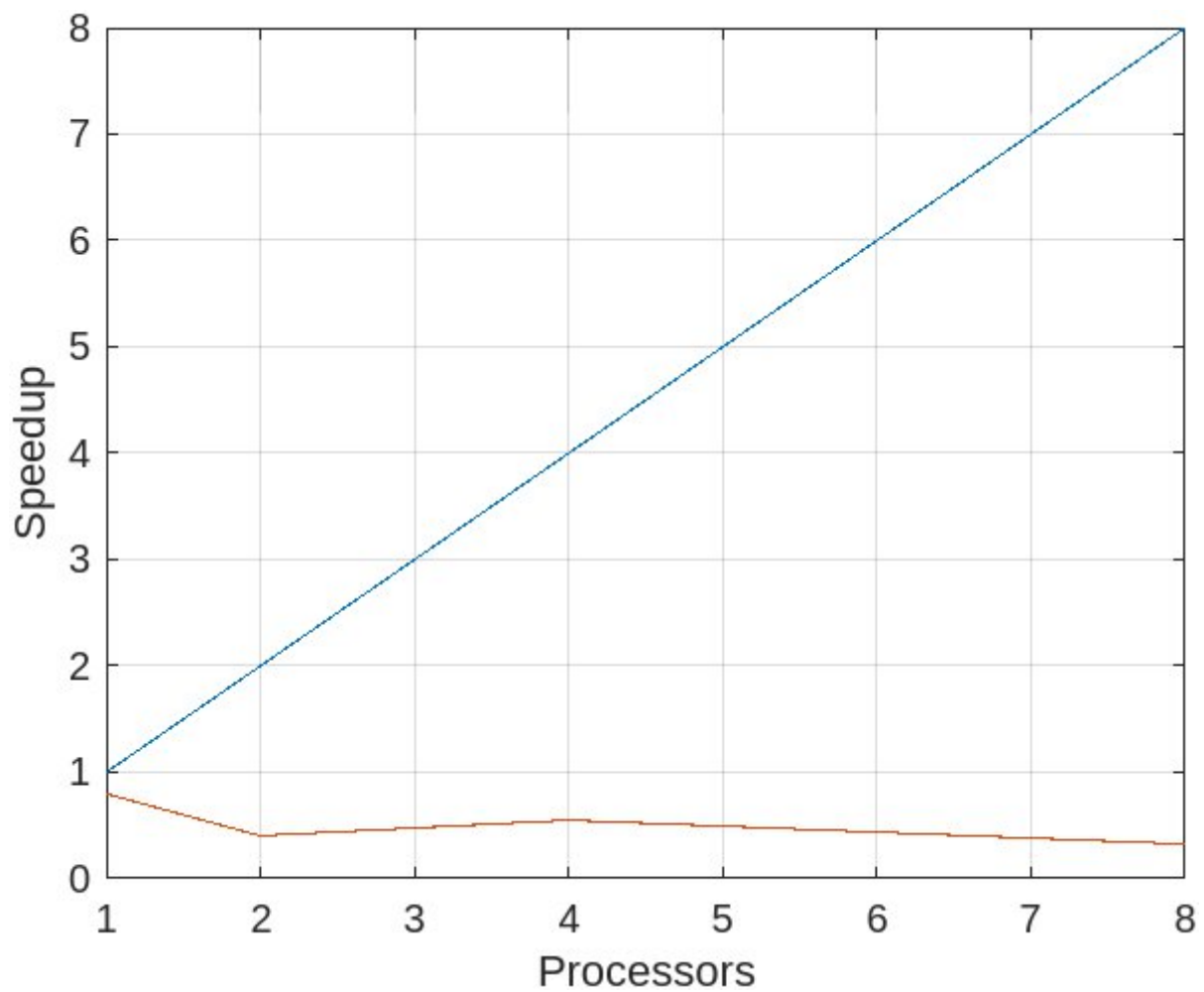
6000 vertices and 250 edges

Version	Processes	Real	User	System	Speedup	Efficiency
Sequential	1	3.754	3.345	0.403	1	1
Parallel	1	8.285	5.658	0.78	0.453108	0.453108
Parallel	2	16.109	21.402	1.871	0.233037	0.116519
Parallel	4	12.065	25.627	4.247	0.311148	0.077787
Parallel	8	20.601	25.597	72.262	0.182224	0.022778



6000 vertices and 400 edges

Version	Processes	Real	User	System	Speedup	Efficiency
Sequential	1	9.233	8.79	0.419	1	1
Parallel	1	11.552	8.8	0.807	0.799256	0.799256
Parallel	2	22.801	32.005	1.957	0.404938	0.202469
Parallel	4	16.775	36.95	4.546	0.550402	0.137601
Parallel	8	28.292	36.799	84.2	0.326347	0.040793



## Final consideration

As expected, we can say that the Tarjan's Algorithm runs much better without parallelization.

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.