UNIVERSITÀ DEGLI STUDI DI SALERNO
Dipartimento di Ingegneria dell'Informazione ed Elettrica e Matematica applicata

Corso di laurea magistrale in Ingegneria Informatica

Parallel version of an algorithm to find the Strongly Connected Components in a Graph

Project Assignment 2022/2023

Lecturer:
Prof. *Francesco Moscato*
fmoscato@unisa.it

Group 04:
Ferrara Grazia 0622701901 g.ferrara75@studenti.unisa.it
Franco Paolo 0622701993 p.franco9@studenti.unisa.it

Contents

Chapter 1

Introduction

1.1 Problem description

The objective of this project is to provide a parallel version of Tarjan's algorithm to find the Strongly Connected Components (SCC) in a Graph. The implementation must use a message-passing paradigm and has to be implemented by using MPI. The input graph needs to be stored on distributed files on each node. In addition also another algorithm, Kosaraju's one, has been evaluated for the same purpose.

1.1.1 About MPI

Message passing interface (MPI) is a standard for communication between nodes belonging to a cluster of computers running a multi-node parallel program and has different implementations. In message-passing programs, a program that runs on a core-memory pair is usually called a process, and two processes can communicate by calling functions: one process calls a send function and the other calls a receive function. MPI creates processes: it is possible to have performance improvements on single/multi core CPUs, despite this, performance does not improve by using more processes than cores. Obviously it is possible to install the MPI middleware to run on multiple nodes in a cluster or even on multiple different nodes on local networks or the internet.

1.2 Solution

The solution to this problem is to split the computation of the strongly connected components between different processes so that each of them computes a piece of the original graph, merges strongly connected components into macro nodes, and communicates with other nodes to gather more pieces of the original graph until the last process manages to compute a graph that is equivalent to the original one.

1.2.1 Main idea under the parallel version

First of all, each process reads its subgraph, a slice of the original graph with a given offset, stored in a binary file named after it, as an array of integers (monodimensional matrix) and inserts it into a SubGraph structure (modeling an adjacency matrix). Then the SubGraph is turned into a ListGraph structure (modeling an adjacency list) to allow more efficient operations on it. After that, each process calculates the SCC of its subgraph and stores this information in another structure called SCCResult. This structure contains an array of linked lists, where each element of the array corresponds to a macro-node of the reduced subgraph, containing all the original nodes defining it. Each process then creates a new graph, rescaling the original graph based on the algorithm's result (replacing nodes with the current suite of

macro-nodes), which will be sent to the next available process together with the updated SCCResult structure.

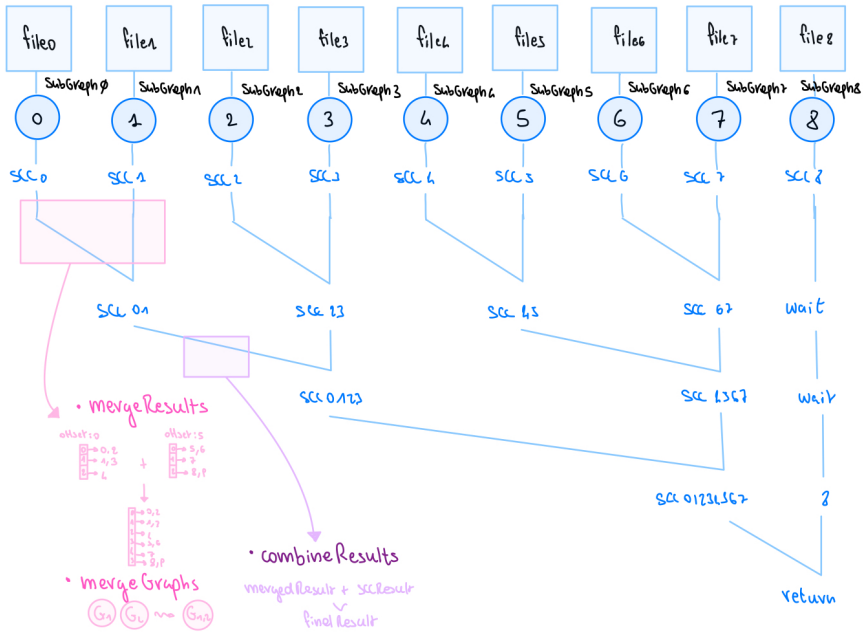

Figure 1.1: Main structure of the parallel algorithm.

The concept on which the communication is based is quite simple: after each iteration of the specific SCC finding algorithm, each process sends its structures to the next active process, which will continue iterating while the first one will terminate its execution. Once a process receives a new subgraph, it merges its original one with the received one, rerouting all the edges for both graphs to keep track of the merged nodes, obtaining a new graph on which it can again recall the chosen algorithm to find SCCs. This series of actions continues until all the original subgraphs have been merged and the final result has reached the last available process.

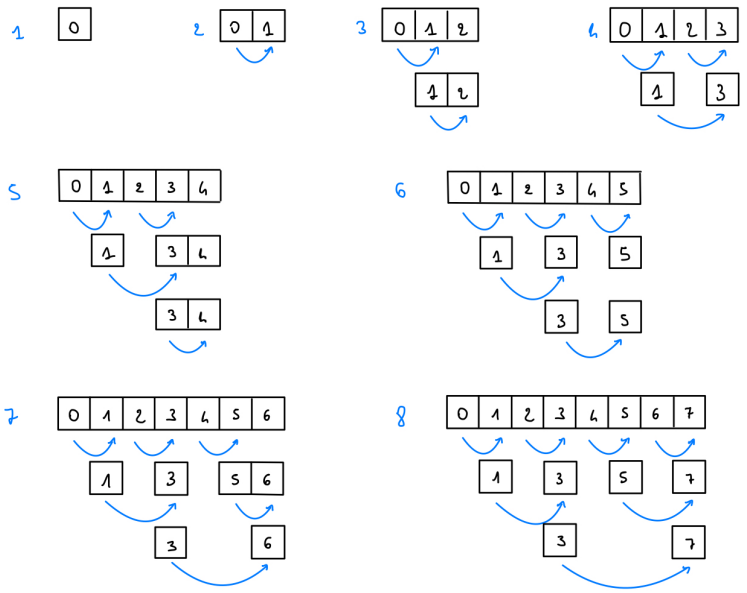

Figure 1.2: Communication among various numbers of processes in the parallel algorithm.

Chapter 2

Custom structures used

In this project, three custom structures were developed, using well-known data structures like stacks, arrays, and unordered lists. The three structures are the *SubGraph*, the *ListGraph*, and the *SCCResutl*.

2.1 SubGraph

The *SubGraph* structure is used to model a subgraph using an adjacency matrix, in which each row and column of the matrix corresponds to a vertex in the graph, and the value at the intersection of a row and column represents the edge between those two vertices. This structure memorizes the number of vertices, nV, of edges, nE, and their offset, needed to understand the position of the subgraph in the original graph. It also memorizes the adjacency matrix, whose dimensions are nV and nE, instantiated as a linear array in which every nE positions correspond to a different row. It is mainly used during the communication, to easily send the whole subgraph in just two communications, the first one to send its dimension and the next one for the whole matrix. When initialized for the first time, after it's been read from the file, its number of vertices will be equals to the chosen workload (number of vertices/number of processes).

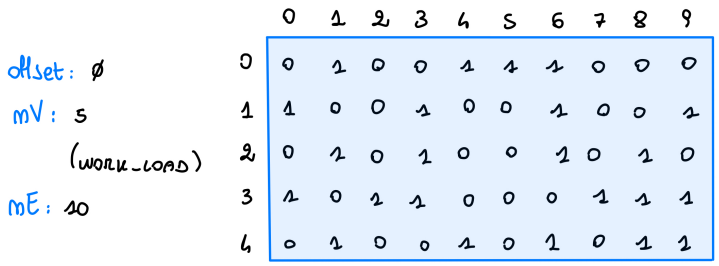

Figure 2.1: The structure of a SubGraph.

2.2 ListGraph

The *ListGraph* structure is used to model a subgraph using an adjacency list representation. First of all, an adjacency list representation is a collection (in this specific case an array) of unordered lists, used to represent a finite graph. The structure contains the number of nodes of the subgraph nV, the number of edges of the subgraph nE, and the offset of the subgraph (used to recognize which part of the full original graph it stores). It also contains the adjacency lists and allows to perform many operations on it such as: create, convert from and to a

SubGraph, print, destroy and insert. While the previous structure has been used for the phases of reading the graphs and communication, this one has been chosen for the operations of SCC computation, rescale and merge, to reduce their complexity and expense.

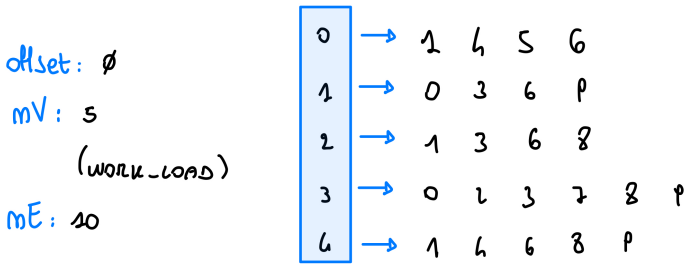

Figure 2.2: The structure of a ListGraph.

2.3 SCCResult

The *SCCResult* structure is used to memorize the composition of the macro nodes for each graph so that in each step of the parallel algorithm it's possible to rebuild the edges of the original subgraph to the new nodes if they have changed during the previous iterations. The structure contains the number of macro nodes, *nMacroNodes*, the offset of the graph, the number of vertices contained in itnV, and an array of linked lists to easily update the content of each macro node.

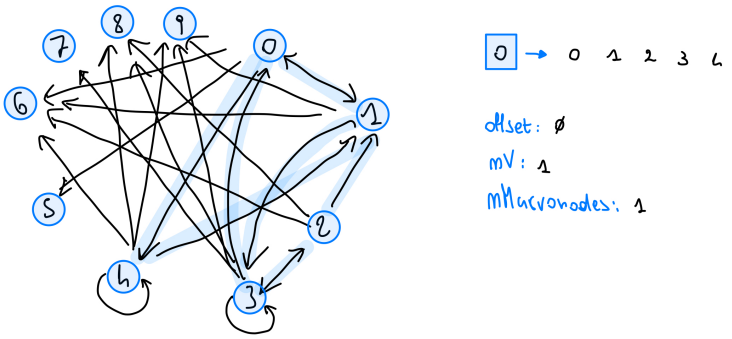

Figure 2.3: The structure of a SccResult relative to the original graph.

Chapter 3

Analysis of the serial algorithm

A strongly connected component (SCC) in a directed graph is a subgraph in which every pair of vertices is mutually reachable, meaning that there is a directed path between any two vertices in the subgraph. A graph can have one or more SCCs and they can be found using several algorithms. Two of them are Tarjan and Kosaraju's ones that guarantee linear execution time.

3.1 Tarjan

Tarjan's algorithm is a graph theory algorithm for finding the strongly connected components of a directed graph. It uses depth-first search to explore the graph and maintains a stack of visited nodes to keep track of the explored path. The algorithm initializes an empty stack and sets all nodes as unvisited. It then performs a depth-first search on each unvisited node, marking it as visited and pushing it onto the stack. As the algorithm explores each node, it keeps track of the lowest "reachable" node it has visited so far, which is defined as the node with the lowest discovery time that can be reached from the current node by following a path of "back edges" (edges that lead from a descendant to an ancestor in the depth-first search tree). When the algorithm reaches a node that has no outgoing edges linking to an unvisited node, it pops all the nodes from the stack until it reaches the node with the lowest reachable node, and these popped nodes form a strongly connected component. The algorithm continues to explore the graph until all nodes have been visited and processed. Tarjan's algorithm is very efficient and has a linear time complexity of $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. It's also used to solve the 2-SAT problem, a particular case of strongly connected components.

3.1.1 Interface

The implementation of the algorithm provided in the project presents the following functions:

- SCC
- SCCUtil

Here is a detailed description of each of them.

SCC

This function finds and inserts in a SCCResult all the strongly connected components found in the graph. It begins by initializing the stack in which to temporarily store all the nodes that belong to the same SCC, the *low* array, used to store the lowest discovery time of a vertex that can be reached from the current vertex following a path of back edges, the *disc* array, used to

keep track of the visited nodes, and the *stackMember* array to quickly check the nodes inserted in the stack. All these structures are required by the auxiliary function *SCCUtil* that is then invoked on each unvisited node of the graph.

```
    SCCResult* SCC( ListGraph** g);
```

Listing 3.1: SCC's interface

SCCUtil

This function is an auxiliary function that is the core of Tarjan's algorithm, it visits the vertex and its neighbors uses a stack to keep track of the current path being explored, and maintains the array of low-link values for each vertex. For each node on whom it's invoked it checks all the adjacent nodes, recursively invoking itself if on the unvisited nodes. If a node has already been visited it compares its low link with that node to the memorized one, updating it if it's lower. Once all the adjacent nodes have been visited it inserts in the *SCCResult* structure all the SCCs that have been found. Its complexity is $O(V + E)$, the same as the whole algorithm itself. The total complexity is kept the same since the function *SCC* recalls this auxiliary function only if the current node is unvisited, so it guarantees to visit exactly one time each node of the graph.

```
void SCCUtil( ListGraph* g, int u, int disc[], int low[],
       TArray* st,int stackMember[],  SCCResult* result);
```

Listing 3.2: SCCUtil's interface

3.2 Kosaraju

Kosaraju's algorithm is another algorithm for finding the strongly connected components of a directed graph. It works by first performing a depth-first search on the graph to create a "finishing time" for each node, which is essentially the order in which the nodes are visited during the search. After the first depth-first search, the algorithm creates a transpose of the original graph, which is a new graph with all the edges reversed. It then performs a second depth-first search on the transpose graph, this time starting with the nodes in the order of their finishing times from the first search. As the second depth-first search is performed, the algorithm keeps track of the strongly connected components by assigning each node to the same component as the first node visited in its connected component during the search. The time complexity of Kosaraju's algorithm is the same as Tarjan's one, so it is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. Kosaraju's algorithm is less efficient than Tarjan's one for arbitrary graphs, as it needs two DFS traversals (one more than Tarjan's).

3.2.1 Interface

The implementation of the algorithm provided in the project presents the following functions:

- SCC_K
- fillOrder
- getTranspose
- DFSUtil

Here is a detailed description of each of them.

SCC_K

This function finds and inserts in a SCCResult all the strongly connected components in the graph. First of all, an array of booleans, called *visited* is created whose size is equal to the number of vertices of the graph. All the elements of the array are then set to false. The following step is to fill the vertices in the stack according to their finishing times and this is achieved through the use of the *fillOrder* function. After that, a reversed graph is obtained using the *getTranspose* function. Subsequently, once again all the elements of the *visited* array are set to false, to perform a second DFS, during which the strongly connected components of each node "popped" from the array, are inserted into the appropriate structure.

```
SCCResult* SCC_K( ListGraph** graph);
```

Listing 3.3: SCC_K's interface

fillOrder

The *fillOrder* recursive function fills the given stack with the vertices in order of their finishing times. First of all the current node (passed as a parameter) is marked to true in the *visited* array, then we loop all over the adjacent vertices of the given node and we recur for each of them. Once all the vertices reachable from v have been processed, we can push v into the stack. Since it is a DFS, its complexity is $O(V + E)$.

```
void fillOrder( ListGraph* g, int v, bool* visited,  TArray* stack);
```

Listing 3.4: fillOrder's interface

getTranspose

The *getTranspose* function is used to obtain the transpose of a given graph. Since in this stage of the algorithm, we are working with the adjacency lists, this operation takes $O(V + E)$ too. We loop all over the vertices of the graph, and for each of them, we recur for all the vertices adjacent to it. In correspondence to each of the adjacent vertices, an insertion into a new ListGraph structure is performed with the node and the adjacent node inverted.

```
ListGraph* getTranspose( ListGraph* g);
```

Listing 3.5: getTranspose's interface

DFSUtil

The *DFSUtil* is a recursive function that, as suggested by its name, performs a DFS, this time on the transposed graph, starting from the node v (popped from the stack) passed as a parameter to the function. This DFS takes $O(V + E)$ too.

```
void DFSUtil( ListGraph* g, int v, bool* visited,  SCCResult* result, int key);
```

Listing 3.6: DFSUtil's interface

Both the algorithm are executed in *serial.c* once the graph has been read through the *fscanf* function from a text file where it has been previously saved from an appropriate script.

Chapter 4

Analysis of the parallel algorithm

The parallel algorithm needs a few more functionalities to work as intended, which are:

- Accessing the SubGraphs
- Communicating the SubGraphs
- Merging the SubGraphs
- Rescaling the SubGraphs
- Combining the SubGraphs

and will be discussed in this chapter. In the end, the expected speedup will also be discussed.

4.1 Accessing the SubGraphs

First of all, each process has to read its own SubGraph from an appropriate file that has previously been produced and named after it.

4.1.1 Generating SubGraphs (*WriteGraph.c*)

SubGraphs are produced by generating random matrices (with fixed offset, minimum and maximum number of edges) and then writing them on files by means of the MPI primitives. Since the opening of a file is a collective primitive (all the processes open the same file) it is necessary, in order to allow them to write on different files at the same moment, to create a temporary communicator for each of them in which they are able to perform separate openings and writings).

```
MPI_Comm file_comm;
MPI_Comm_split(MPI_COMM_WORLD, rank, rank, &file_comm);
MPI_File_open(file_comm, filename, MPI_MODE_CREATE | MPI_MODE_RDWR,
    MPI_INFO_NULL, &fh);
MPI_File_seek(fh, WORK_LOAD*WORK_LOAD*size, MPI_SEEK_SET);
MPI_File_write(fh, edges, WORK_LOAD*WORK_LOAD*size, MPI_INT, &status);
MPI_File_close(&fh);
MPI_Comm_free(&file_comm);
```

Listing 4.1: The code that manages file writing

The code above is the one used to write the subgraphs on the files. The *WORK_LOAD* constant, as suggested by its name, represents the amount of workload (number of vertices of the original graph for each process), while *size* is the total number of available processes. It is now possible to understand that each produced SubGraph has *WORK_LOAD* vertices and *size* * *WORK_LOAD* edges.

4.1.2 Reading SubGraphs (*Parallel.c*)

The readings are made similarly, using the *MPI_File_read* primitive, instead of the one which performs the write.

```
int MPI_File_read(MPI_File fh, void* buf, int count,
    MPI_Datatype datatype, MPI_Status* status)
```

Listing 4.2: MPI_File_read's interface

In both cases, the file is opened in the *MPI_MODE_CREATE | MPI_MODE_RDWR* modality, which creates the file if it does not exist yet, and overwrites it otherwise. Since the matrix is represented in the SubGraph structure as a monodimensional array, it is both read and written into a buffer of *MPI_INT*.

4.2 Communicating the SubGraphs

As said before the concept on which the communication is based is quite simple. More specifically the concept is that after each iteration of the algorithm, each even-ranked process sends its structures to the next odd-ranked process, which will continue iterating while the even-ranked one will terminate its execution. To obtain this behavior all the processes have an internal array that represents the current processes' situation. At the beginning of the program, all the processes generate this boolean array setting all the items to *true*, then, iteration by iteration, each process updates its local array simulating what happened after the last communication. Each process will update each even-indexed *true* item to *false* so that in the next iteration it will be possible to deduce its role by looking into this array. It could happen that given certain numbers of active processes, the last one of them is unable to find the next available node; in this case, it will behave as if it has already executed this iteration and will continue with the next one.

4.2.1 Sending function (*Communication.c*)

The function that manages to send all the structures is the *send_all* function.

```
void send_all(SubGraph* graph, SCCResult* result,int shrink, int dest)
```

Listing 4.3: send_all's interface

This function sends all the structures to the *dest* process, by invoking three different *MPI_Send*. The first one is used to send the dimensions of the graph, the offset, and the *shrink* value so that the receiver can prepare the structures required to perform the communication. During the second communication, a $nV*(nE+1)$ matrix is sent; it is made of the adjacency matrix of the graph and, for each vertex, the length of the linked list associated with that macro node. All the linked lists of the *SCCResult* structure are sent in the last communication, through an array whose dimension equals the sum of all the lengths previously sent.

4.2.2 Reciving function (*Communication.c*)

The function that manages to receive all the structures is the *recv_all* function.

```
void recv_all(SubGraph** graph, SCCResult** result, int* shrink, int source)
```

Listing 4.4: recv_all's interface

This function receives all the structures from the *source* process, by invoking three different *MPI_Recv*, similarly to what is done in the *send_all* function. The order in which the structures are received is the same since both *MPI_Send* and *MPI_Recv* implement synchronous direct communication over the MPI protocol.

4.3 Merging the SubGraphs (*Merge.c*)

Once the communication has taken place between two processes, the received SubGraph has to be converted into a ListGraph, on which it is possible to execute many expensive operations. Then it is necessary to perform two different kinds of merge in the following order:

1. **Merging the SCCResults**. Given the two results (the one received from the previously available process and its own), the receiving process has to merge the two structures to obtain a new result which is the concatenation of the previous two. The new structure has an array whose size is the sum of the other two and whose linked lists are the ones of the other two structures reported sequentially.

```
SCCResult *mergeResults(SCCResult *r1, SCCResult *r2);
```

Listing 4.5: mergeResults's interface

2. **Merging the ListGraphs**. Given the two *ListGraphs* to be merged, their respective shrinks (the number of nodes they have lost in the last SCC computation) and the *merged* SCCResult. First of all, an array mapping the nodes (indices of the array) to the macro nodes of the *merged* structure is created, to make all the following operations less expensive. Then we iterate all over the vertices of the first ListGraph, inserting the appropriate nodes (or macronodes if they have been merged during the computing of the SCCs) into the new ListGraph representing the merged subgraph. After this, it's the turn of the second structure and the process is repeated, taking into account the differences in terms of offset and shrink.

```
ListGraph *mergeGraphs(ListGraph *g1, ListGraph *g2, int shrink1, int shrink2,
    SCCResult *merged);
```

Listing 4.6: mergeGraphs's interface

4.4 Rescaling the SubGraphs (*Tarjan.c*)

Once the SCC finding algorithm has been executed, the input *SubGraph* needs to be modified to merge all the SCCs in macro nodes and reroute all the edges. This is done by the *rescaleGraph* function, whose signature is the following:

```
ListGraph *rescaleGraph(ListGraph** oldGraph, SCCResult* tarjan)
```

Listing 4.7: rescaleGraph's interface

This function uses a dynamic programming approach to have a time complexity equal to $O(V+E)$. It begins by instantiating an array of size nV that will be used as a map to track the original position of a node to its new one, reflecting what is stored inside the *SCCResult* structure. Then it cycles through all the input *SubGraph* rerouting all the edges with the value contained in the array precedently built. In the end, it also manages to destroy the input *SubGraph* so that it can no longer be used.

4.5 Combining the SubGraphs (*SCCResult.c*)

Once the graph has been rescaled there is just one step to do before the start of the next iteration. The original *SubGraph* must be updated to represent the rescaled graph. This is accomplished by the *SCCResultCombine* function, whose signature is the following:

```
SCCResult *SCCResultCombine(SCCResult *tarjanResult, SCCResult *mergedSCC)
```

Listing 4.8: SCCResultCombine's interface

In this function, a new *SCCResult* is created, with a size equal to Tarjan's result. Then it iterates through all the nodes contained in this structure, copying the linked lists of the visited nodes in their new positions.

4.6 Expected speedup

Both Tarjan's and Kosaraju's algorithms for finding the strongly connected components in a graph have a time complexity C_{serial} of $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. This means that the algorithm's running time is directly proportional to the size of the input. Parallelizing this algorithm it's possible to lower the dimensions of the graph on which the algorithm is executed, expecting faster executions at each step. By using n processors that compute different parts of the graph it's possible to expect an execution n times faster at each iteration, but the need to reiterate the algorithm until the whole original graph has been fully analyzed increases the required computation of a factor equals to *logn*, that represents the number of required communications.

4.6.1 Efficiency and speedup

To better quantify these factors it's useful defining the *speedup* of an algorithm, which is a measure of relative performances between two systems or algorithms that execute the same problem. It can be measured as the ratio between serial and parallel complexity or time. Similar to the concept of speedup there is the concept of *efficiency* of a parallel algorithm, this is a measure to detect the quality of an algorithm and can be calculated as:

$$E_r = \frac{T_1}{P \cdot T_P} \tag{4.1}$$

in which:

- T_1 is the time to execute an algorithm on just one processor
- P is the number of used processors
- T_P is the time to execute an algorithm in parallel on P processors

At this point, it's also possible to find a relationship between *relative speedup* and *relative efficiency*:

$$S_r = P \cdot E_r \tag{4.2}$$

Going back to the analysis of the parallel algorithm, it's possible to state that in the average case, when each process manages to shrink the input subgraph, the theoretical speedup should be of a factor:

$$S_{average} = \frac{n}{log_2 n} \tag{4.3}$$

However in the worst case, when there are no SCCs in the graph, at each step the algorithm doesn't find any nodes to regroup, leaving the subgraph unchanged, which leads to an actual decrease in the speedup: at each step, all the active processors will have to analyze bigger and bigger subgraphs only not to find anything. In this case, the total time complexity can be calculated from the formula:

$$C_{parallel} = C_{serial} * \sum_{i=0}^{\log_2 n} \frac{n}{2^i} = C_{serial} * \frac{2n-1}{n} \tag{4.4}$$

That brings the theoretical speedup, calculated as serial complexity divided by parallel one, to a factor equal to:

$$S_{worst\ case} = \frac{n}{2n-1} \in \{0,1\}\, \forall\, n \in N \tag{4.5}$$

However consider that all the parallel speedups were calculated without considering the time and complexity needed for the communication and the merging of the received data, nor the speedup due to the parallel reading of the graph. So the effective speedup could be lower or even higher than the theoretical one, depending on the specific situation in which it's executed.

Chapter 5

Experimental setup

Both the serial and parallel programs were tested on a Raspberry cluster and a MacBook Pro.

5.1 Cluster

The cluster is composed of a front-end Raspberry pi3 and four back-end Raspberry pi 4, using a distributed NFS file system. The front-end device has a Cortex-A53 processor, based on the ARMv8-A microarchitecture, with these main specifications:

- 8-stage pipelined processor with 2-way superscalar
- In-order execution pipeline
- DSP and NEON SIMD extensions for each core core
- VFPv4 Floating Point Unit onboard for each core
- Hardware virtualization support and TrustZone security extensions
- 64-byte cache lines
- 8-64 KiB of L1 cache
- 128 KiB-2 MiB of L2 cache
- 10-entry and 512-entry TLBs respectively for L1 and L2 cache
- Conditional branch predictor of 4 KiB, together with a 256-entry indirect branch predictor

The back-end devices have a Cortex-A72 processor, based on the same microarchitecture of the front-end device but with a series of improvements. Its main specifications are:

- 15-stage pipelined processor with 3-way superscalar execution pipeline
- Deeply out-of-order execution pipeline
- DSP and NEON SIMD extensions for each core core
- VFPv4 Floating Point Unit onboard for each core
- Hardware virtualization support and TrustZone security extensions
- Supports Thumb-2 instruction set encoding to reduce the size of 32-bit programs with little impact on performance.

- Program Trace Macrocell and CoreSight Design Kit for unobtrusive tracing of instruction execution
- 32 - 48 KiB of L1 cache
- 512 KB-4 MB of L2 cache
- 64-entry and 1024-entry TLBs respectively for L1 and L2 cache, with support for hit-under-miss
- Sophisticated branch prediction algorithm that significantly increases performance and reduces energy from misprediction and speculation
- Regionalized TLB and μ BTB tagging
- Suppression of superfluous branch predictor accesses

5.2 MacBook

The MacBook Pro (13-inch, 2019, Two Thunderbolt 3 ports)'s main specifications are:

- Processor name: Intel Core i5 quad-core
- Processor velocity: 1,4 GHz
- Number of processors: 1
- Total number of cores: 4
- Cache L2 (per Core): 256 KB
- Cache L3: 6 MB
- Hyper-Threading technology: Allowed
- Memory: 8 GB

Chapter 6

Performance, speedup and efficiency

6.1 Tarjan

In this section are reported the measures and the results in terms of speedup and efficiency. We have chosen to test Tarjan's algorithm with different sizes of the graphs (400, 800, 1200, 1600, 2000, 2400) and for each of them to consider different amounts of edges-for-vertex: first with all the possible edges for each vertex (shrink 1), then with just the half of the maximum number (shrink 2) and last with a quarter of them (shrink 4). We will refer to the number of edges we are considering as shrink. First, we report the measures made with the cluster and then the ones made with the laptop. Note that the speedup measured on the cluster is relative to the difference between the front-end execution and the back-end one, where there are newer and faster devices, so the speedup here has to be considered very relative.

6.1.1 SIZE-400 Cluster

With the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	1.67121	1.31974	2.65762	1.0	1.0
Parallel	1	1.66706	2.17831	3.84537	0.69112	0.69112
Parallel	2	1.13488	1.78441	2.9193	0.91036	0.45518
Parallel	4	1.35125	1.57531	2.65168	1.00224	0.25056
Parallel	8	0.6886	0.89998	1.58858	1.67295	0.20912
Parallel	16	0.53819	0.71769	1.25588	2.11614	0.13226

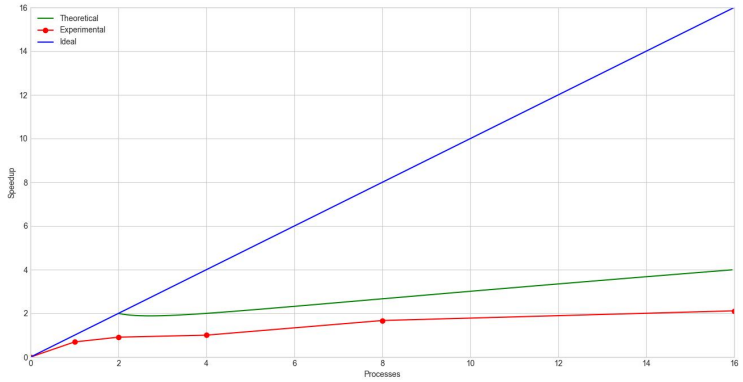

Figure 6.1: Speedup of the 400 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	0.52422	0.63505	1.15927	1.0	1.0
Parallel	1	0.40878	0.55907	0.96785	1.19778	1.19778
Parallel	2	0.31035	0.83747	1.14782	1.00998	0.50499
Parallel	4	0.5727	1.14941	1.72211	0.67317	0.16829
Parallel	8	1.42506	1.11625	2.5413	0.45617	0.05702
Parallel	16	0.5779	1.0295	1.60741	0.7212	0.04508

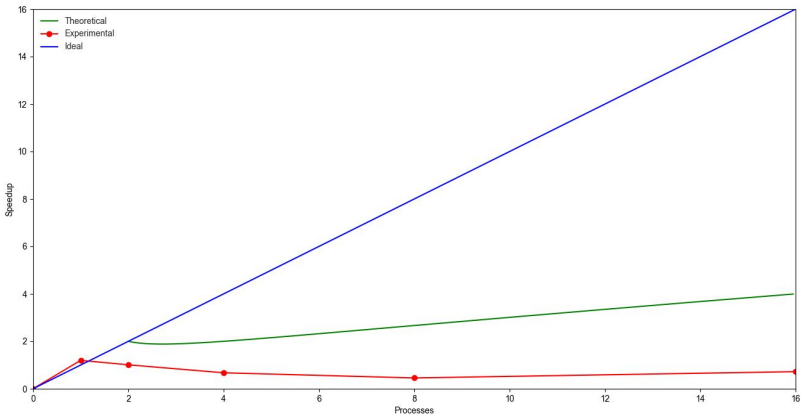

Figure 6.2: Speedup of the 400 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	0.22262	0.10272	0.32534	1.0	1.0
Parallel	1	0.22874	0.06937	0.29812	1.09131	1.09131
Parallel	2	0.38627	0.39956	0.78583	0.41401	0.207
Parallel	4	0.37869	0.65034	1.02903	0.31616	0.07904
Parallel	8	0.42601	1.0138	1.43981	0.22596	0.02825
Parallel	16	0.66681	0.83989	1.5067	0.21593	0.0135

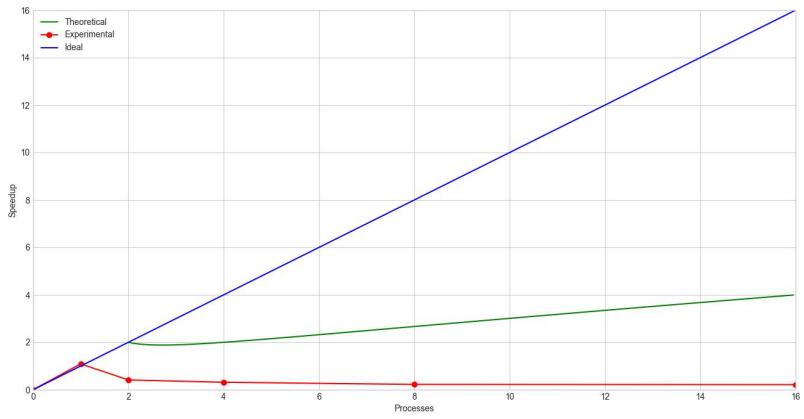

Figure 6.3: Speedup of the 400 vertices graph with shrink 4.

6.1.2 SIZE-800 Cluster

With the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	8.43781	12.16297	20.60078	1.0	1.0
Parallel	1	8.13	15.74144	23.87144	0.86299	0.86299
Parallel	2	3.65535	9.95752	13.61287	1.51333	0.75667
Parallel	4	1.29107	7.63969	8.93076	2.30672	0.57668
Parallel	8	3.60509	1.39581	5.00091	4.11941	0.51493
Parallel	16	2.5592	0.57423	3.13343	6.57451	0.41091

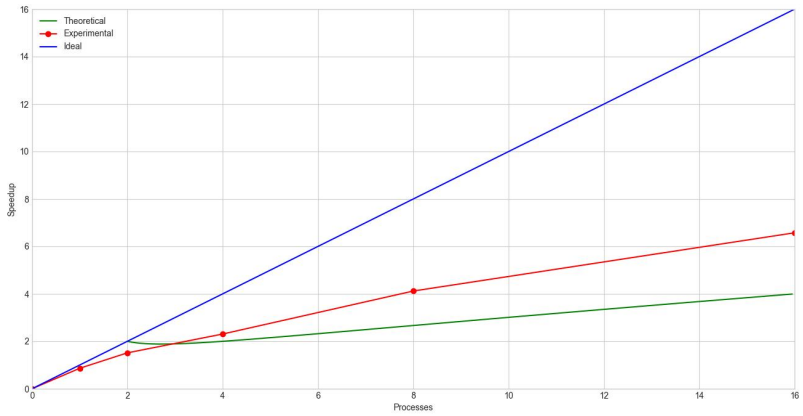

Figure 6.4: Speedup of the 800 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	2.37439	2.82885	5.20323	1.0	1.0
Parallel	1	2.2007	3.58912	5.78982	0.89869	0.89869
Parallel	2	2.46421	5.9366	8.40081	0.61937	0.30969
Parallel	4	2.20195	8.32243	10.52438	0.4944	0.1236
Parallel	8	1.19595	7.27609	8.47204	0.61416	0.07677
Parallel	16	0.94936	5.57597	6.52534	0.79739	0.04984

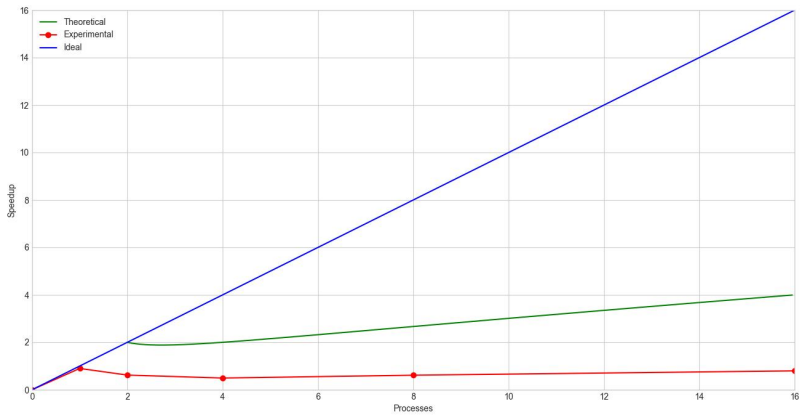

Figure 6.5: Speedup of the 800 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	1.2552	0.74309	1.99828	1.0	1.0	
	Parallel	1	0.72079	1.23105	1.95184	1.02379	1.02379	
	Parallel	2	1.10776	2.31793	3.42569	0.58332	0.29166	
	Parallel	4	1.81736	3.63394	5.45129	0.36657	0.09164	
	Parallel	8	0.84585	4.70306	5.54891	0.36012	0.04502	
	Parallel	16	0.63504	3.85906	4.4941	0.44465	0.02779	

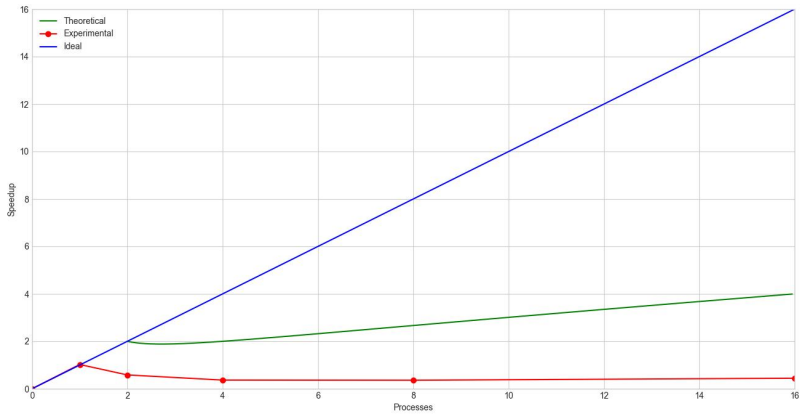

Figure 6.6: Speedup of the 800 vertices graph with shrink 4.

6.1.3 SIZE-1200 Cluster

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	25.75535	37.36818	63.12353	1.0	1.0	
	Parallel	1	10.74086	22.22805	32.9689	1.91464	1.91464	
	Parallel	2	5.62894	14.93222	20.56116	3.07004	1.53502	
	Parallel	4	3.90976	11.30219	15.21195	4.1496	1.0374	
	Parallel	8	5.95969	8.13223	14.09192	4.47941	0.55993	
	Parallel	16	1.72929	8.10032	9.82961	6.42177	0.40136	

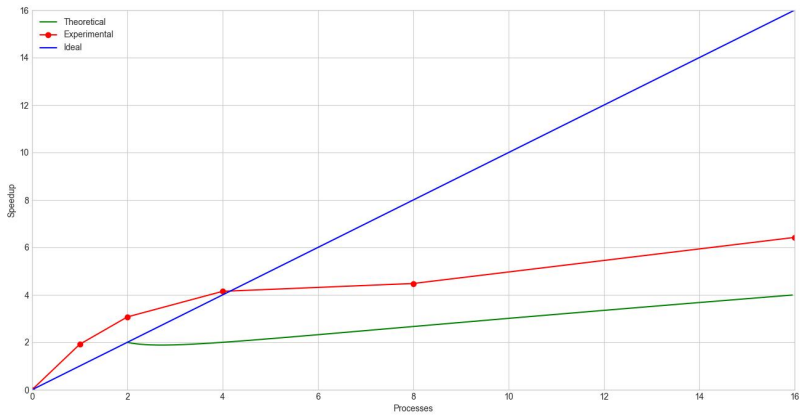

Figure 6.7: Speedup of the 1200 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	7.51263	9.36094	16.87357	1.0	1.0
Parallel	1	6.56286	11.83893	18.40179	0.91695	0.91695
Parallel	2	6.8638	17.78399	24.64778	0.68459	0.34229
Parallel	4	6.86336	28.46957	35.33293	0.47756	0.11939
Parallel	8	3.63287	23.06275	26.69561	0.63207	0.07901
Parallel	16	2.11598	16.73538	18.85136	0.89509	0.05594

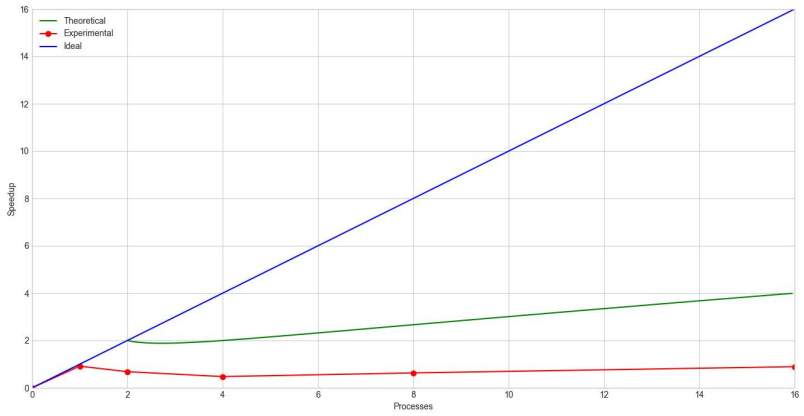

Figure 6.8: Speedup of the 1200 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	2.73101	2.4195	5.15051	1.0	1.0
Parallel	1	1.76981	4.12573	5.89554	0.87363	0.87363
Parallel	2	3.32728	8.44888	11.77616	0.43737	0.21868
Parallel	4	2.57267	11.29239	13.86507	0.37147	0.09287
Parallel	8	1.87783	19.21632	21.09415	0.24417	0.03052
Parallel	16	1.03134	11.98821	13.01955	0.3956	0.02472

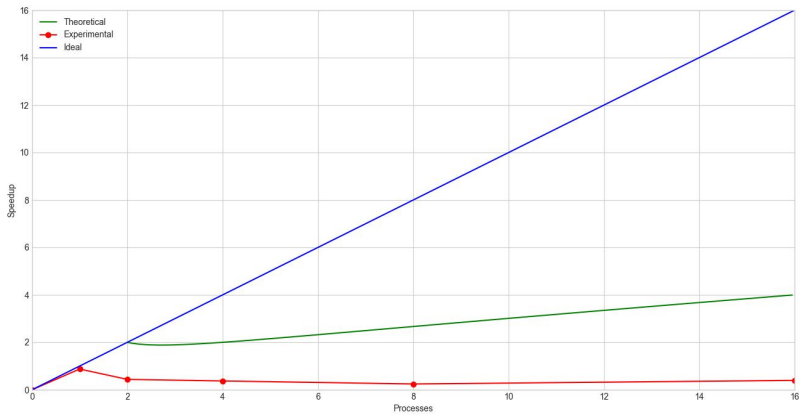

Figure 6.9: Speedup of the 1200 vertices graph with shrink 4.

6.1.4 SIZE-1600 Cluster

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	60.05434	90.82746	150.8818	1.0	1.0	
	Parallel	1	24.96649	56.23393	81.20042	1.85814	1.85814	
	Parallel	2	12.96192	36.79249	49.7544	3.03253	1.51627	
	Parallel	4	8.97355	28.14037	37.11392	4.06537	1.01634	
	Parallel	8	5.05244	49.12406	54.1765	2.785	0.34813	
	Parallel	16	3.23955	45.15325	48.3928	3.11786	0.19487	

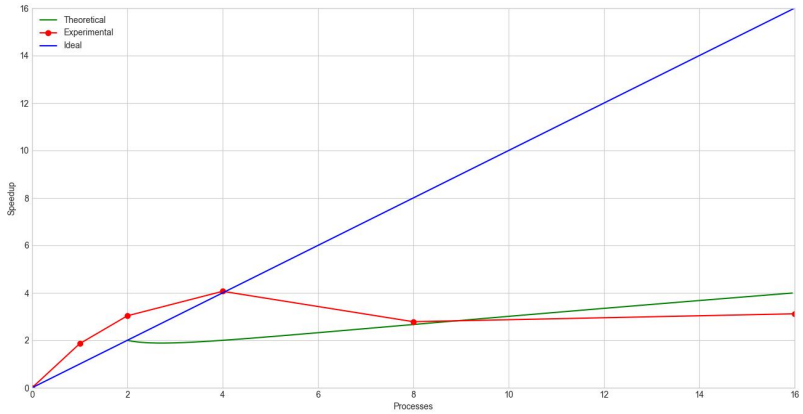

Figure 6.10: Speedup of the 1600 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	16.56085	22.03295	38.59381	1.0	1.0	
	Parallel	1	16.01982	31.50797	47.52779	0.81203	0.81203	
	Parallel	2	16.42307	48.19608	64.61916	0.59725	0.29863	
	Parallel	4	18.0067	60.67993	78.68663	0.49047	0.12262	
	Parallel	8	8.33371	53.43229	61.766	0.62484	0.0781	
	Parallel	16	4.0129	46.37105	50.38395	0.76599	0.04787	

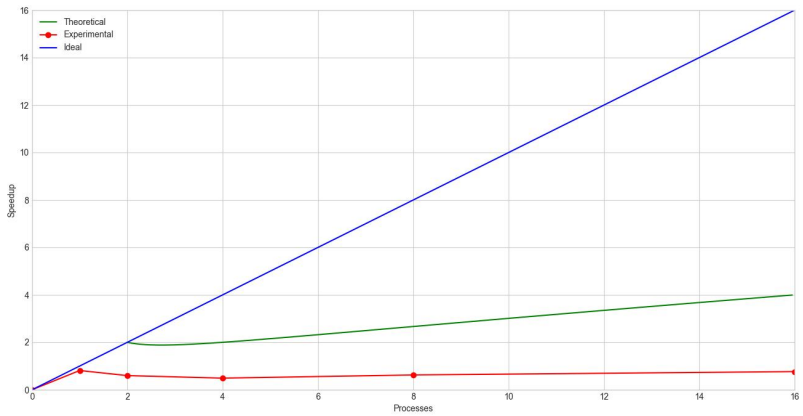

Figure 6.11: Speedup of the 1600 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	6.49234	5.66435	12.15669	1.0	1.0
Parallel	1	6.00804	11.63175	17.63979	0.68916	0.68916
Parallel	2	6.45456	16.2551	22.70967	0.53531	0.26765
Parallel	4	6.00787	26.63085	32.63872	0.37246	0.09312
Parallel	8	3.76604	22.79248	26.55852	0.45773	0.05722
Parallel	16	1.75093	20.35095	22.10187	0.55003	0.03438

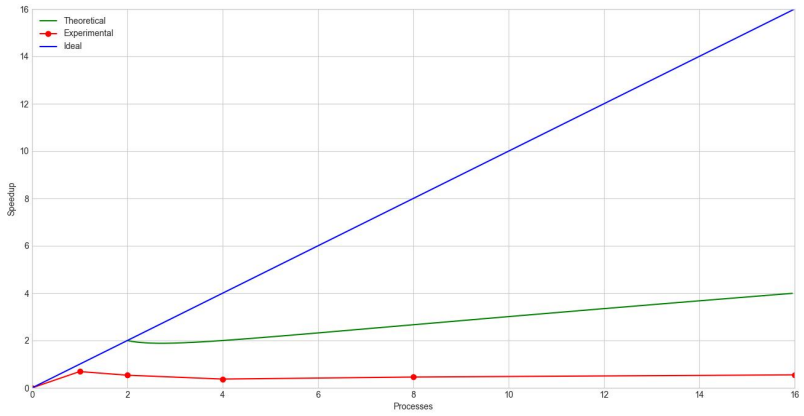

Figure 6.12: Speedup of the 1600 vertices graph with shrink 4.

6.1.5 SIZE-2000 Cluster

With the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	116.94591	176.67774	293.62366	1.0	1.0
Parallel	1	48.32631	111.82317	160.14948	1.83343	1.83343
Parallel	2	24.93957	73.10204	98.04162	2.99489	1.49744
Parallel	4	18.85767	60.80916	79.66683	3.68565	0.92141
Parallel	8	9.29325	95.28851	104.58175	2.8076	0.35095
Parallel	16	6.68958	78.66532	85.3549	3.44003	0.215

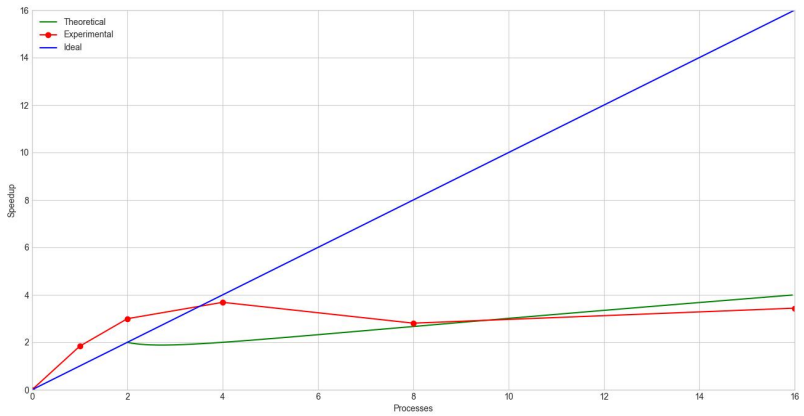

Figure 6.13: Speedup of the 2000 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	56.94591	66.67774	122.95699	1.0	1.0
Parallel	1	48.32631	50.82317	99.14948	1.24012	1.24012
Parallel	2	68.32631	70.82317	138.14948	0.89003	0.44501
Parallel	4	48.50905	49.40864	97.58435	1.26001	0.315
Parallel	8	48.50905	39.40864	87.58435	1.40387	0.17548
Parallel	16	48.50905	39.40864	77.58435	1.58482	0.09905

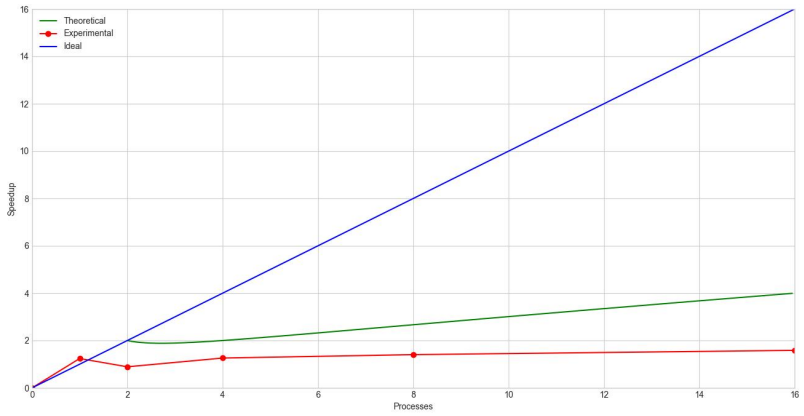

Figure 6.14: Speedup of the 2000 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	10.83704	11.45425	22.29128	1.0	1.0
Parallel	1	11.47804	18.56825	30.0463	0.7419	0.7419
Parallel	2	11.6537	28.92713	40.58083	0.54931	0.27465
Parallel	4	11.50905	49.40864	60.91769	0.36592	0.09148
Parallel	8	5.9299	40.10882	46.03872	0.48419	0.06052
Parallel	16	2.93284	44.01147	46.9443	0.47485	0.02968

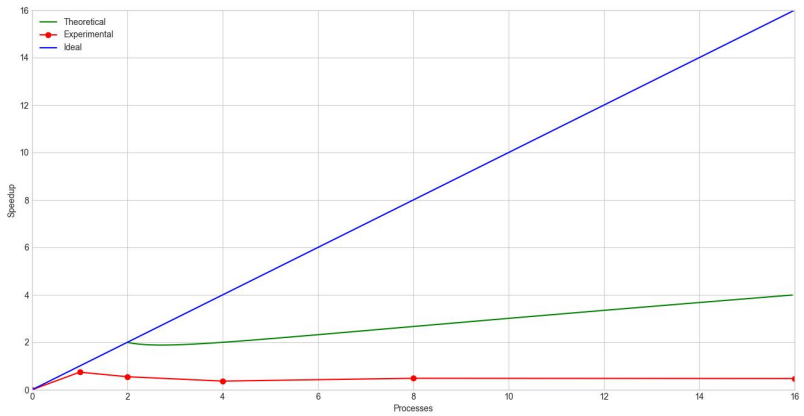

Figure 6.15: Speedup of the 2000 vertices graph with shrink 4.

6.1.6 SIZE-400 PC

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.02235	0.02233	0.04468	1.0	1.0	
	Parallel	1	0.01052	0.02215	0.03267	1.36762	1.36762	
	Parallel	2	0.00553	0.03417	0.0397	1.12544	0.56272	
	Parallel	4	0.00323	0.02239	0.02562	1.74395	0.43599	
	Parallel	8	0.08787	0.06421	0.15208	0.29379	0.03672	

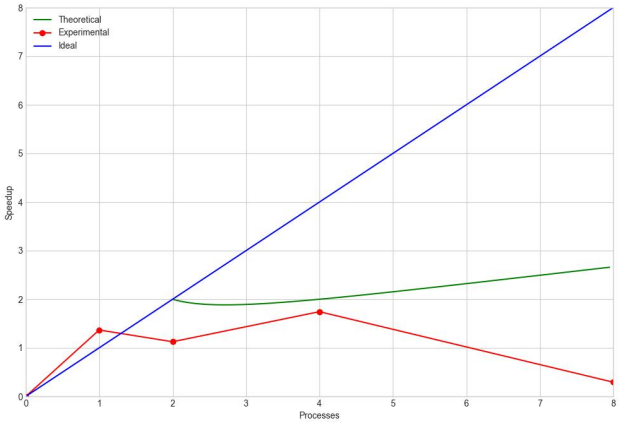

Figure 6.16: Speedup of the 400 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.01668	0.01091	0.02759	1.0	1.0	
	Parallel	1	0.00655	0.01103	0.01758	1.5694	1.5694	
	Parallel	2	0.00378	0.04003	0.04381	0.62976	0.31488	
	Parallel	4	0.00258	0.02303	0.0256	1.07773	0.26943	
	Parallel	8	0.08414	0.06574	0.14988	0.18408	0.02301	

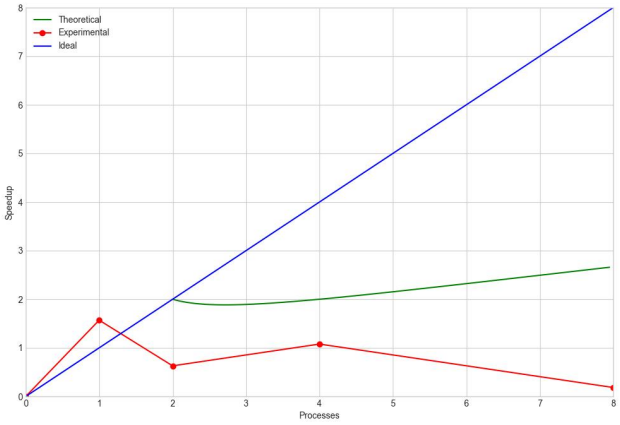

Figure 6.17: Speedup of the 400 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.01378	0.00558	0.01936	1.0	1.0	
	Parallel	1	0.00414	0.00552	0.00966	2.00414	2.00414	
	Parallel	2	0.00221	0.01625	0.01846	1.04875	0.52438	
	Parallel	4	0.00132	0.01735	0.01868	1.0364	0.2591	
	Parallel	8	0.05705	0.04556	0.10261	0.18868	0.02358	

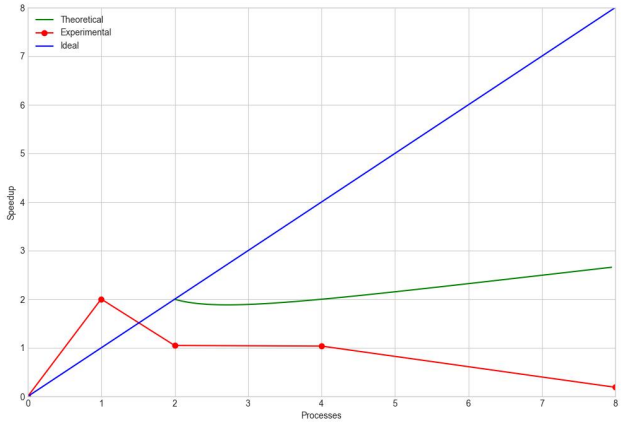

Figure 6.18: Speedup of the 400 vertices graph with shrink 4.

6.1.7 SIZE-800 PC

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.08146	0.08792	0.16938	1.0	1.0	
	Parallel	1	0.04105	0.09097	0.13203	1.28289	1.28289	
	Parallel	2	0.02053	0.11011	0.13064	1.29654	0.64827	
	Parallel	4	0.01163	0.09436	0.10599	1.59808	0.39952	
	Parallel	8	0.08092	0.14868	0.2296	0.73772	0.09221	

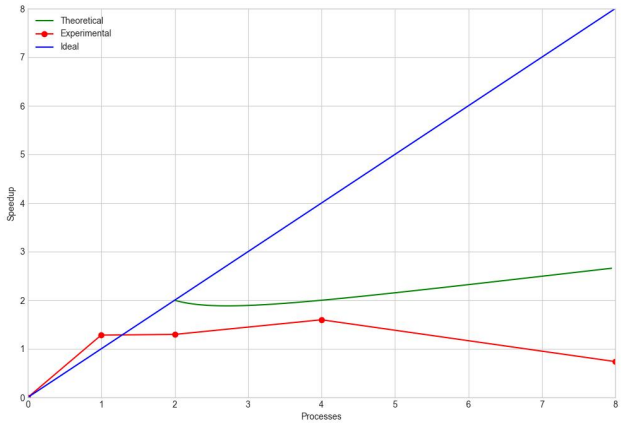

Figure 6.19: Speedup of the 800 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	0.06639	0.04463	0.11102	1.0	1.0
Parallel	1	0.02599	0.04478	0.07076	1.56897	1.56897
Parallel	2	0.01351	0.07646	0.08997	1.23397	0.61698
Parallel	4	0.00692	0.09159	0.09851	1.12699	0.28175
Parallel	8	0.06911	0.19171	0.26082	0.42566	0.05321

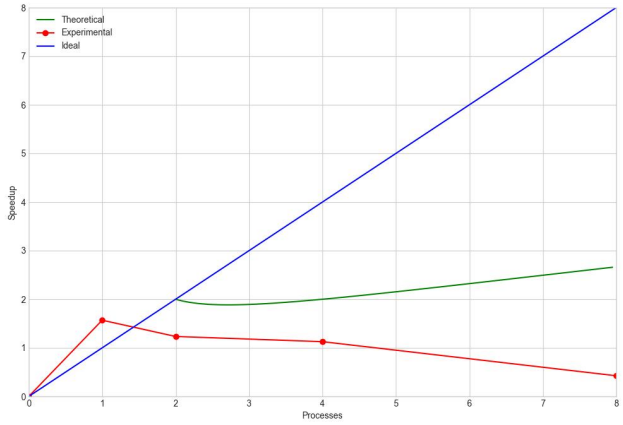

Figure 6.20: Speedup of the 800 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	0.05522	0.02245	0.07767	1.0	1.0
Parallel	1	0.01623	0.02405	0.04028	1.92825	1.92825
Parallel	2	0.00797	0.0484	0.05637	1.37786	0.68893
Parallel	4	0.00476	0.05736	0.06212	1.25032	0.31258
Parallel	8	0.08985	0.17576	0.26561	0.29242	0.03655

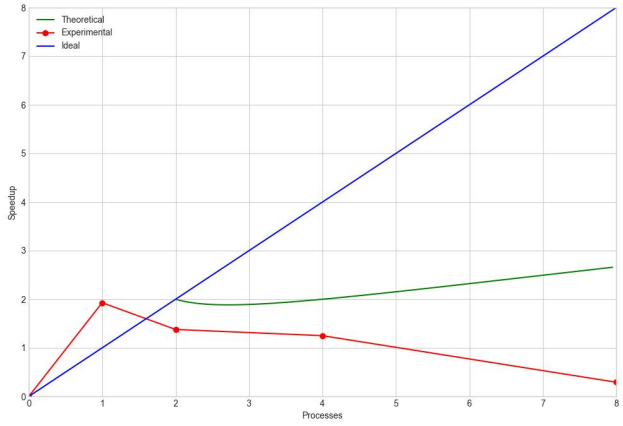

Figure 6.21: Speedup of the 800 vertices graph with shrink 4.

6.1.8 SIZE-1200 PC

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.183	0.20219	0.38519	1.0	1.0	
	Parallel	1	0.09195	0.20467	0.29662	1.2986	1.2986	
	Parallel	2	0.05073	0.27147	0.3222	1.1955	0.59775	
	Parallel	4	0.02403	0.20973	0.23376	1.6478	0.41195	
	Parallel	8	0.07517	0.30925	0.38441	1.00203	0.12525	

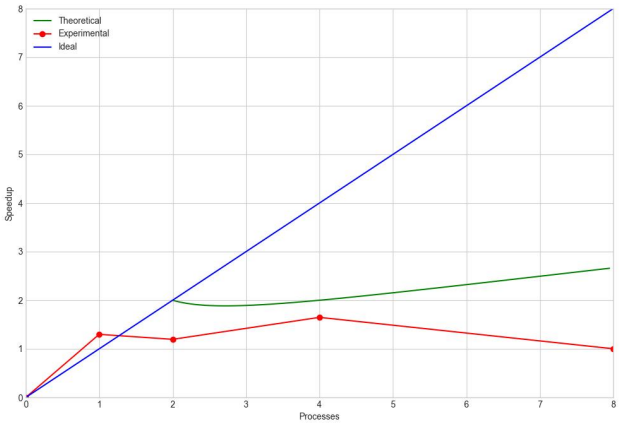

Figure 6.22: Speedup of the 1200 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.14941	0.1	0.24941	1.0	1.0	
	Parallel	1	0.05814	0.10081	0.15895	1.56911	1.56911	
	Parallel	2	0.03011	0.16973	0.19984	1.24805	0.62402	
	Parallel	4	0.01578	0.20056	0.21634	1.15286	0.28822	
	Parallel	8	0.12214	0.43304	0.55519	0.44923	0.05615	

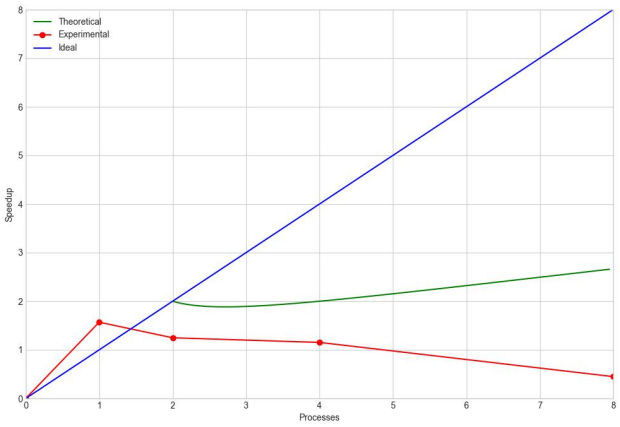

Figure 6.23: Speedup of the 1200 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.12409	0.05041	0.17451	1.0	1.0	
	Parallel	1	0.03526	0.05158	0.08684	2.00956	2.00956	
	Parallel	2	0.01746	0.10748	0.12495	1.39664	0.69832	
	Parallel	4	0.01169	0.13631	0.148	1.17912	0.29478	
	Parallel	8	0.092	0.33501	0.427	0.40869	0.05109	

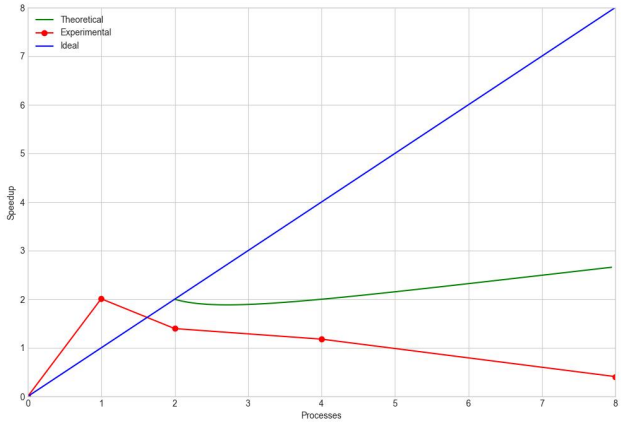

Figure 6.24: Speedup of the 1200 vertices graph with shrink 4.

6.1.9 SIZE-1600 PC

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.32444	0.35321	0.67765	1.0	1.0	
	Parallel	1	0.16305	0.35462	0.51767	1.30904	1.30904	
	Parallel	2	0.08974	0.49121	0.58095	1.16645	0.58323	
	Parallel	4	0.04479	0.40722	0.45201	1.49919	0.3748	
	Parallel	8	0.11034	0.56384	0.67417	1.00516	0.12565	

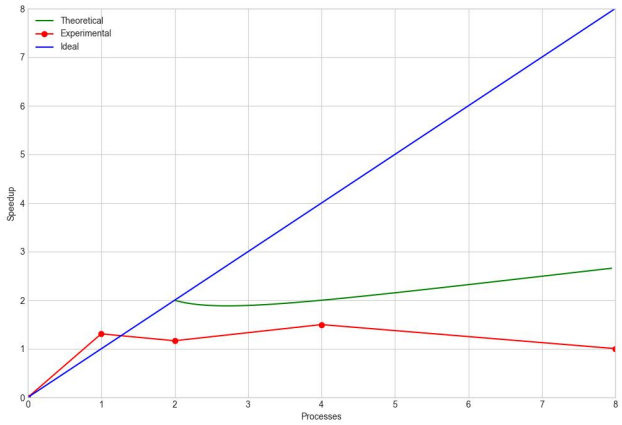

Figure 6.25: Speedup of the 1600 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	0.26345	0.17732	0.44078	1.0	1.0
Parallel	1	0.10372	0.18045	0.28417	1.55111	1.55111
Parallel	2	0.05186	0.30456	0.35642	1.23669	0.61834
Parallel	4	0.02828	0.34813	0.37641	1.17101	0.29275
Parallel	8	0.10212	0.75842	0.86054	0.51221	0.06403

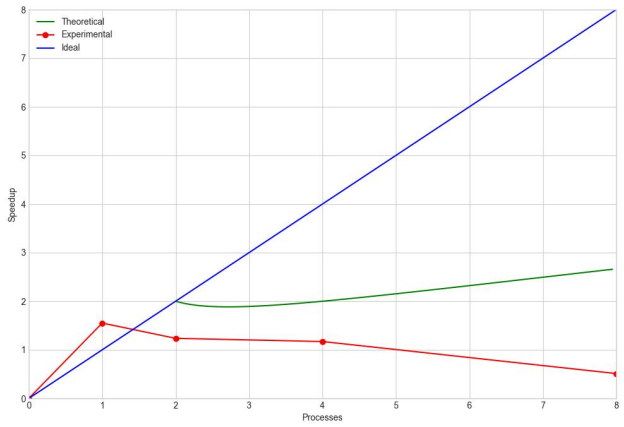

Figure 6.26: Speedup of the 1600 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	0.22144	0.08806	0.3095	1.0	1.0
Parallel	1	0.06492	0.09293	0.15785	1.96072	1.96072
Parallel	2	0.0316	0.19072	0.22233	1.39207	0.69604
Parallel	4	0.01803	0.24493	0.26296	1.17699	0.29425
Parallel	8	0.07118	0.56934	0.64051	0.48321	0.0604

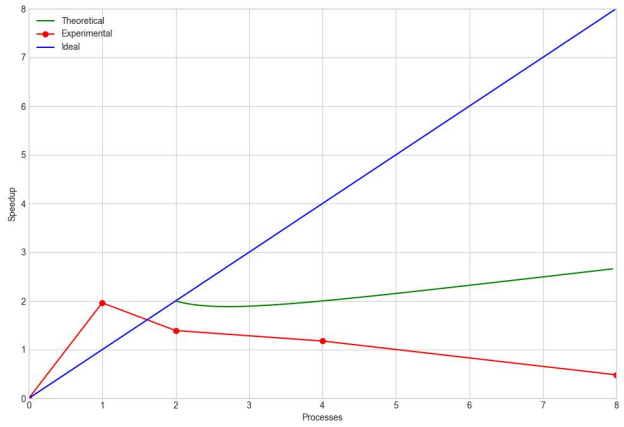

Figure 6.27: Speedup of the 1600 vertices graph with shrink 4.

6.1.10 SIZE-2000 PC

With the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	0.50831	0.55042	1.05873	1.0	1.0
Parallel	1	0.25366	0.55431	0.80797	1.31036	1.31036
Parallel	2	0.12676	0.67117	0.79793	1.32685	0.66342
Parallel	4	0.06831	0.5945	0.66281	1.59734	0.39933
Parallel	8	0.15664	0.85637	1.01301	1.04513	0.13064

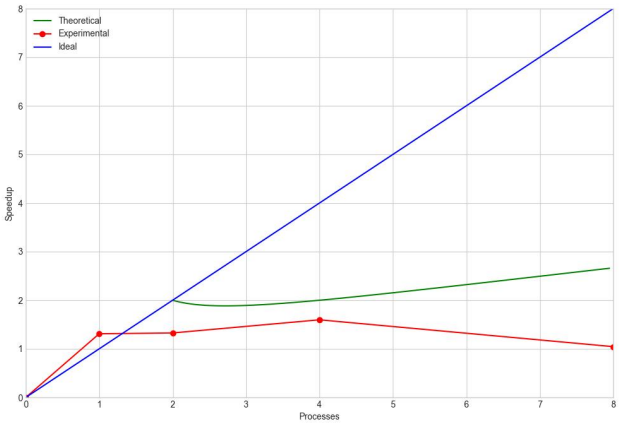

Figure 6.28: Speedup of the 2000 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	0.41133	0.27288	0.68421	1.0	1.0
Parallel	1	0.16506	0.27675	0.44181	1.54865	1.54865
Parallel	2	0.08088	0.45903	0.53991	1.26727	0.63363
Parallel	4	0.0502	0.63201	0.68222	1.00292	0.25073
Parallel	8	0.11369	1.14435	1.25804	0.54387	0.06798

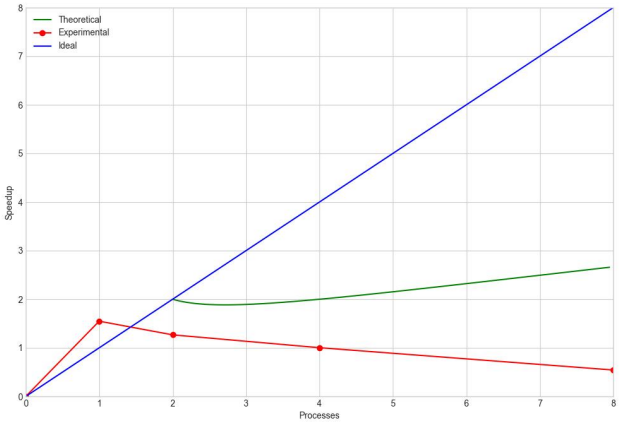

Figure 6.29: Speedup of the 2000 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.34472	0.14163	0.48635	1.0	1.0	
	Parallel	1	0.09662	0.14213	0.23876	2.03698	2.03698	
	Parallel	2	0.04835	0.28262	0.33098	1.46942	0.73471	
	Parallel	4	0.02773	0.36811	0.39585	1.22862	0.30716	
	Parallel	8	0.09526	0.84632	0.94159	0.51652	0.06456	

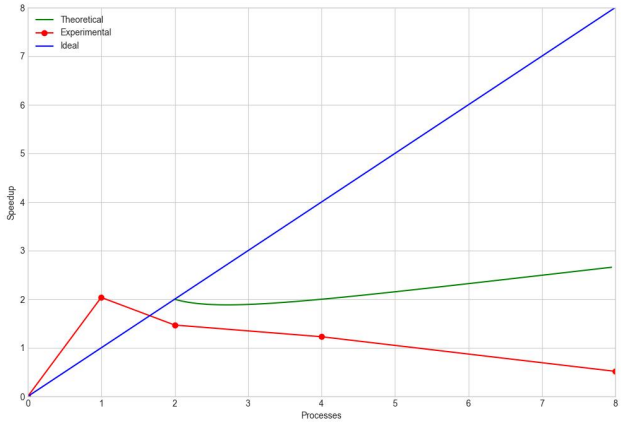

Figure 6.30: Speedup of the 2000 vertices graph with shrink 4.

6.1.11 SIZE-2400 PC

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.73006	0.78397	1.51403	1.0	1.0	
	Parallel	1	0.36817	0.79118	1.15935	1.30593	1.30593	
	Parallel	2	0.18314	0.95683	1.13998	1.32812	0.66406	
	Parallel	4	0.10088	0.84962	0.9505	1.59288	0.39822	
	Parallel	8	0.19815	1.33711	1.53526	0.98617	0.12327	

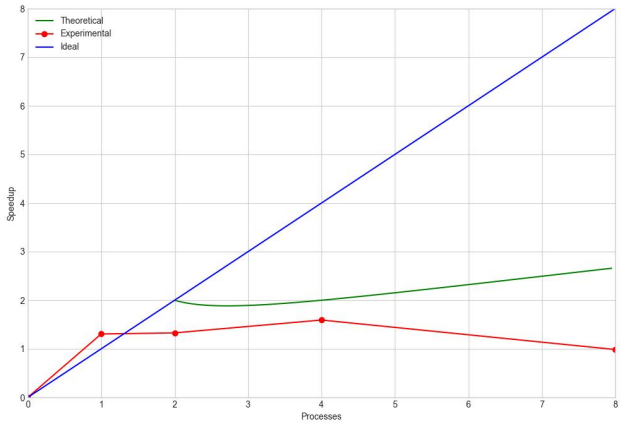

Figure 6.31: Speedup of the 2400 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	0.59456	0.39226	0.98682	1.0	1.0
Parallel	1	0.23282	0.39735	0.63017	1.56596	1.56596
Parallel	2	0.11804	0.65386	0.7719	1.27843	0.63921
Parallel	4	0.06198	0.80369	0.86567	1.13995	0.28499
Parallel	8	0.14773	1.65452	1.80225	0.54755	0.06844

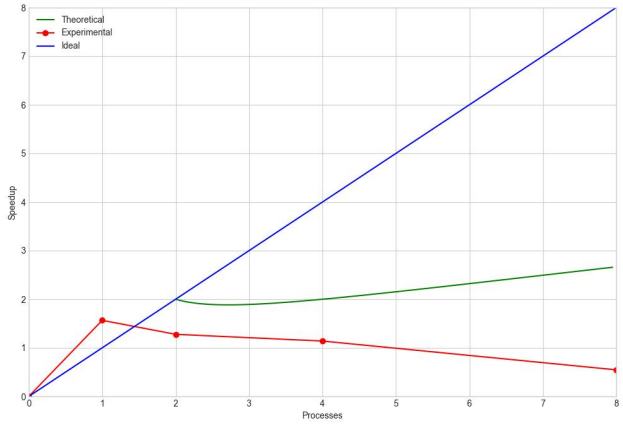

Figure 6.32: Speedup of the 2400 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	0.49543	0.20031	0.69575	1.0	1.0
Parallel	1	0.13913	0.20004	0.33917	2.05133	2.05133
Parallel	2	0.06976	0.38297	0.45273	1.53679	0.76839
Parallel	4	0.03734	0.52091	0.55825	1.24631	0.31158
Parallel	8	0.10309	1.09452	1.19761	0.58095	0.07262

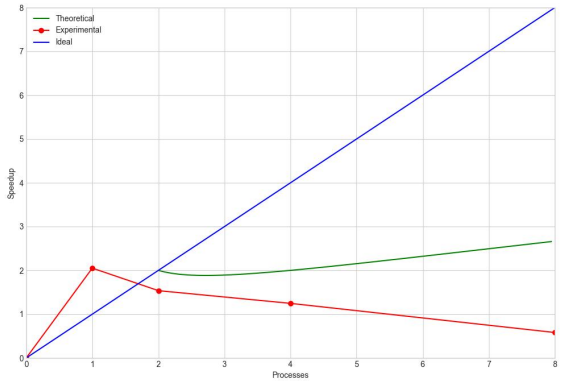

Figure 6.33: Speedup of the 2400 vertices graph with shrink 4.

6.2 Kosaraju

The same approach is now applied to the Kosaraju's algorithm.

6.2.1 SIZE-400 PC

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.02019	0.03327	0.05346	1.0	1.0	
	Parallel	1	0.01033	0.03362	0.04395	1.21638	1.21638	
	Parallel	2	0.00538	0.03668	0.04205	1.27134	0.63567	
	Parallel	4	0.00281	0.03078	0.03359	1.59155	0.39789	
	Parallel	8	0.0549	0.09651	0.15141	0.35308	0.04414	

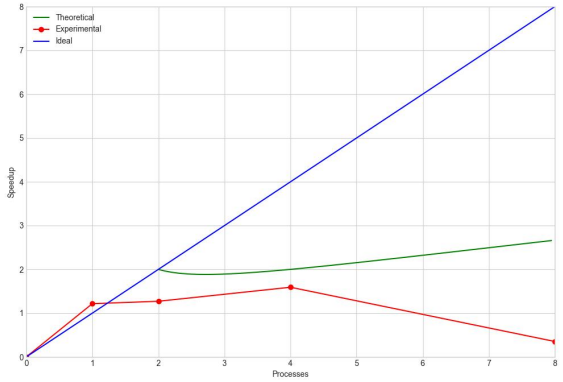

Figure 6.34: Speedup of the 400 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.01645	0.01627	0.03272	1.0	1.0	
	Parallel	1	0.00668	0.01618	0.02286	1.43132	1.43132	
	Parallel	2	0.00353	0.0457	0.04923	0.66464	0.33232	
	Parallel	4	0.00227	0.02793	0.0302	1.08344	0.27086	
	Parallel	8	0.04281	0.09476	0.13757	0.23784	0.02973	

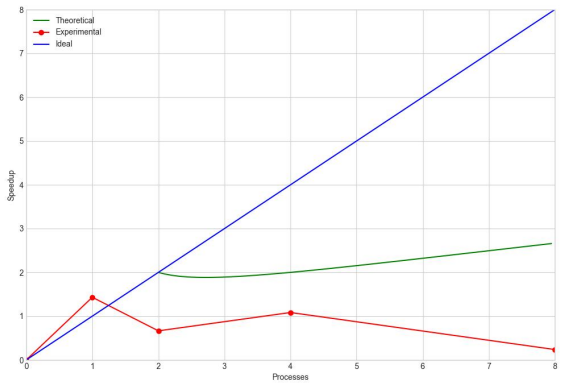

Figure 6.35: Speedup of the 400 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.01381	0.00792	0.02174	1.0	1.0	
	Parallel	1	0.00408	0.0081	0.01218	1.78489	1.78489	
	Parallel	2	0.00218	0.01599	0.01817	1.19648	0.59824	
	Parallel	4	0.00134	0.02197	0.0233	0.93305	0.23326	
	Parallel	8	0.05	0.04871	0.09871	0.22024	0.02753	

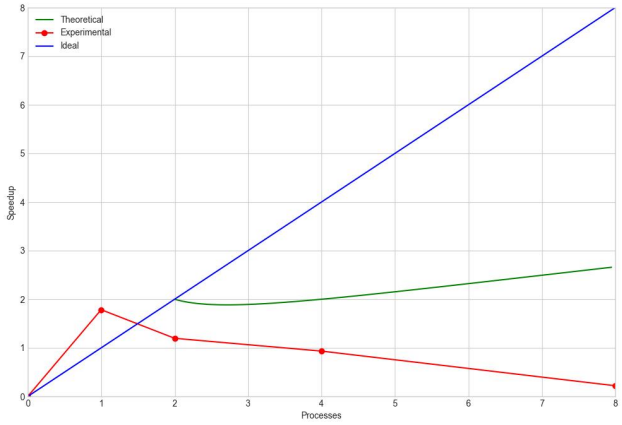

Figure 6.36: Speedup of the 400 vertices graph with shrink 4.

6.2.2 SIZE-800 PC

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.08226	0.13387	0.21613	1.0	1.0	
	Parallel	1	0.04193	0.13692	0.17884	1.20851	1.20851	
	Parallel	2	0.02084	0.14766	0.16851	1.28259	0.6413	
	Parallel	4	0.01065	0.12765	0.1383	1.56276	0.39069	
	Parallel	8	0.06743	0.20587	0.2733	0.79082	0.09885	

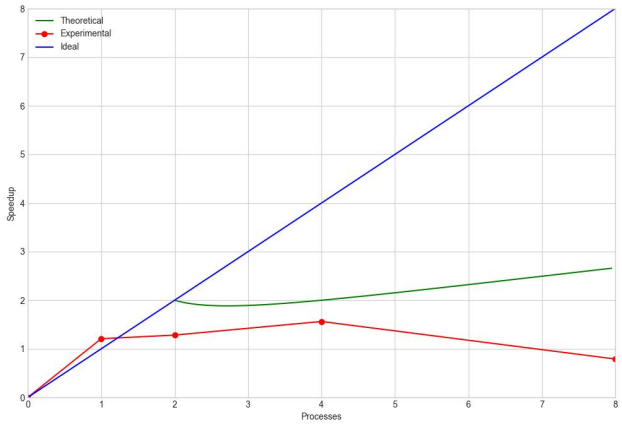

Figure 6.37: Speedup of the 800 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.06611	0.0667	0.13281	1.0	1.0	
	Parallel	1	0.02608	0.06934	0.09542	1.39185	1.39185	
	Parallel	2	0.01329	0.09846	0.11175	1.18846	0.59423	
	Parallel	4	0.00719	0.12517	0.13236	1.0034	0.25085	
	Parallel	8	0.07768	0.22018	0.29786	0.44588	0.05574	

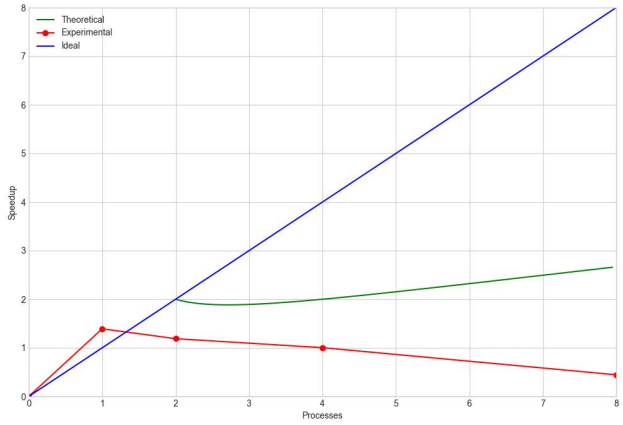

Figure 6.38: Speedup of the 800 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.05559	0.03436	0.08995	1.0	1.0	
	Parallel	1	0.0159	0.03547	0.05137	1.75102	1.75102	
	Parallel	2	0.00805	0.06395	0.072	1.24931	0.62465	
	Parallel	4	0.00418	0.07004	0.07422	1.21194	0.30298	
	Parallel	8	0.05279	0.19027	0.24307	0.37006	0.04626	

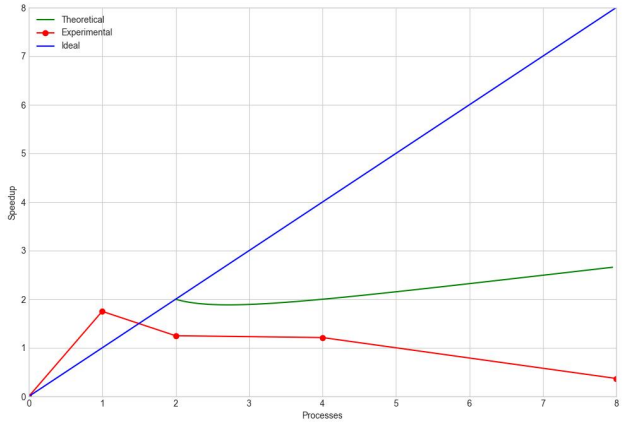

Figure 6.39: Speedup of the 800 vertices graph with shrink 4.

6.2.3 SIZE-1200 PC

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.18227	0.29593	0.47819	1.0	1.0	
	Parallel	1	0.09148	0.30235	0.39383	1.2142	1.2142	
	Parallel	2	0.04737	0.37184	0.4192	1.14072	0.57036	
	Parallel	4	0.02569	0.30389	0.32958	1.45091	0.36273	
	Parallel	8	0.09117	0.40024	0.49141	0.9731	0.12164	

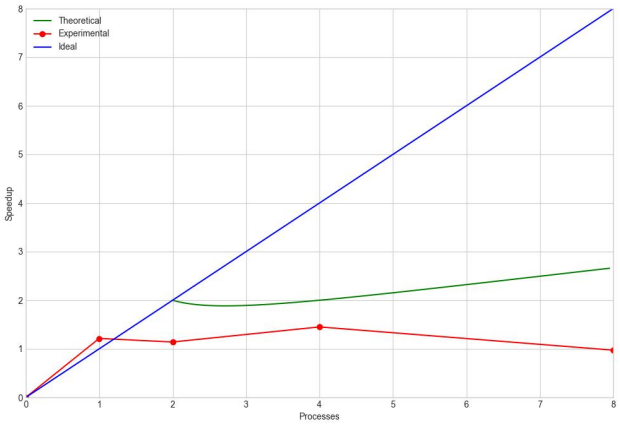

Figure 6.40: Speedup of the 1200 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.14888	0.15008	0.29896	1.0	1.0	
	Parallel	1	0.05856	0.15396	0.21252	1.40674	1.40674	
	Parallel	2	0.02993	0.22301	0.25294	1.18194	0.59097	
	Parallel	4	0.01579	0.24754	0.26332	1.13535	0.28384	
	Parallel	8	0.06888	0.53779	0.60667	0.49279	0.0616	

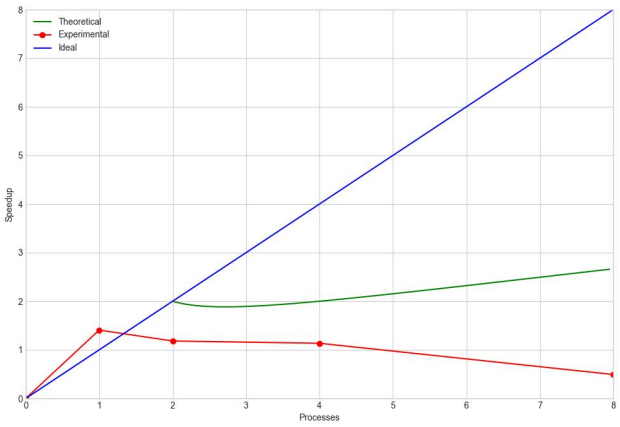

Figure 6.41: Speedup of the 1200 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.12382	0.0771	0.20092	1.0	1.0	
	Parallel	1	0.03509	0.07798	0.11307	1.77695	1.77695	
	Parallel	2	0.01761	0.13316	0.15077	1.33263	0.66631	
	Parallel	4	0.00953	0.16512	0.17465	1.15042	0.2876	
	Parallel	8	0.08896	0.43052	0.51948	0.38677	0.04835	

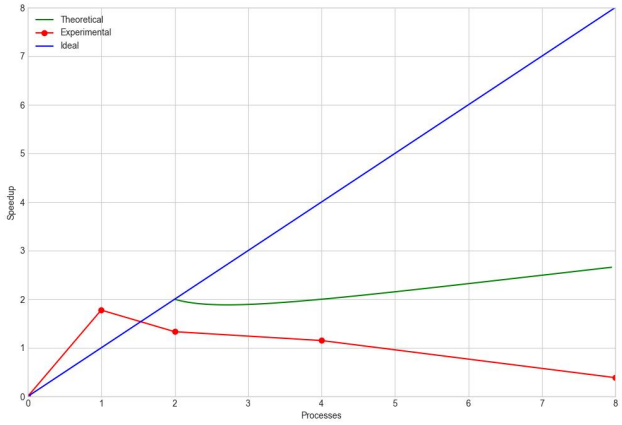

Figure 6.42: Speedup of the 1200 vertices graph with shrink 4.

6.2.4 SIZE-1600 PC

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.32756	0.52788	0.85544	1.0	1.0	
	Parallel	1	0.16311	0.53958	0.70269	1.21738	1.21738	
	Parallel	2	0.08599	0.66457	0.75056	1.13974	0.56987	
	Parallel	4	0.04514	0.54218	0.58732	1.45651	0.36413	
	Parallel	8	0.12688	0.75415	0.88102	0.97097	0.12137	

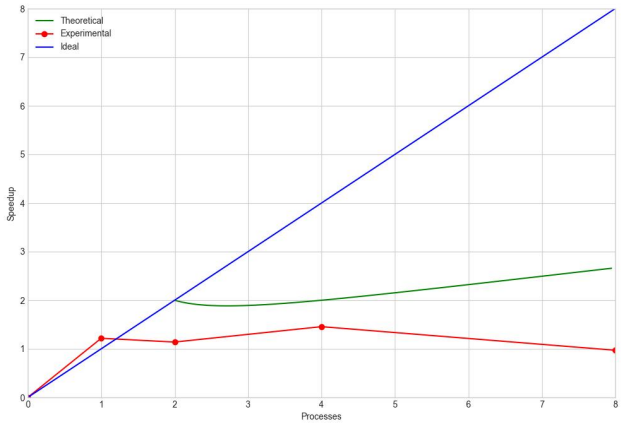

Figure 6.43: Speedup of the 1600 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.26416	0.26619	0.53035	1.0	1.0	
	Parallel	1	0.10417	0.26854	0.37272	1.42292	1.42292	
	Parallel	2	0.05192	0.38272	0.43465	1.22018	0.61009	
	Parallel	4	0.02902	0.41227	0.44129	1.20182	0.30045	
	Parallel	8	0.08064	0.92873	1.00938	0.52542	0.06568	

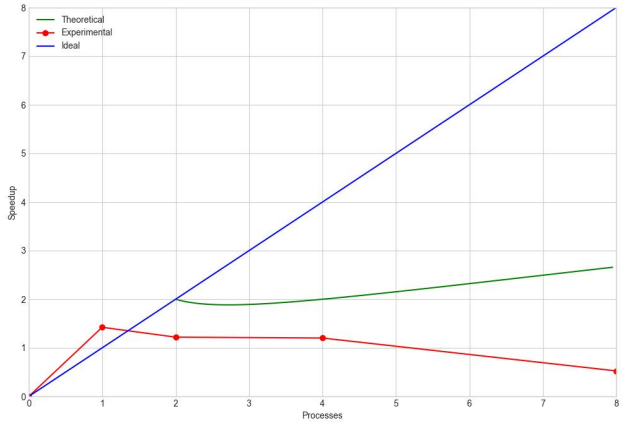

Figure 6.44: Speedup of the 1600 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.22079	0.13529	0.35609	1.0	1.0	
	Parallel	1	0.06312	0.1409	0.20402	1.74537	1.74537	
	Parallel	2	0.03185	0.22847	0.26032	1.36789	0.68395	
	Parallel	4	0.01784	0.30254	0.32038	1.11146	0.27787	
	Parallel	8	0.07693	0.6687	0.74563	0.47757	0.0597	

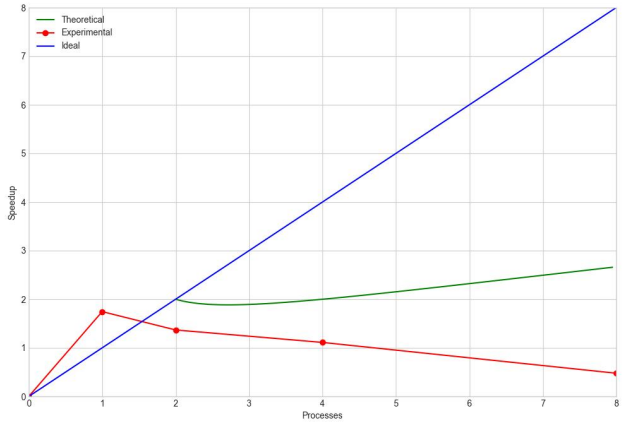

Figure 6.45: Speedup of the 1600 vertices graph with shrink 4.

6.2.5 SIZE-2000 PC

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.50798	0.83354	1.34152	1.0	1.0	
	Parallel	1	0.25616	0.84683	1.10299	1.21626	1.21626	
	Parallel	2	0.12766	0.92632	1.05398	1.27281	0.63641	
	Parallel	4	0.06761	0.82028	0.88789	1.51091	0.37773	
	Parallel	8	0.13593	1.14937	1.2853	1.04374	0.13047	

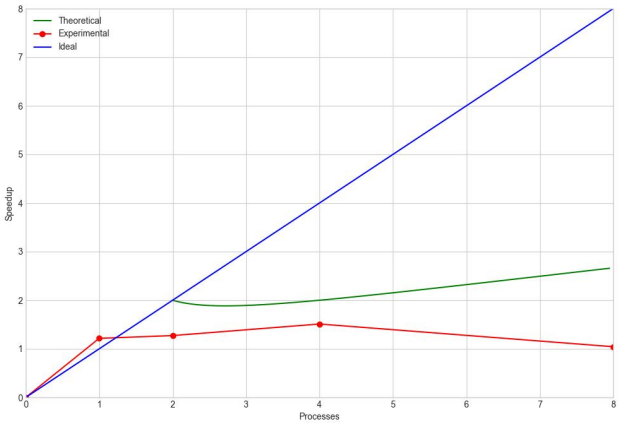

Figure 6.46: Speedup of the 2000 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.41237	0.41518	0.82755	1.0	1.0	
	Parallel	1	0.16257	0.42578	0.58835	1.40656	1.40656	
	Parallel	2	0.08132	0.58995	0.67128	1.23279	0.6164	
	Parallel	4	0.04641	0.79103	0.83744	0.98819	0.24705	
	Parallel	8	0.09586	1.47221	1.56807	0.52775	0.06597	

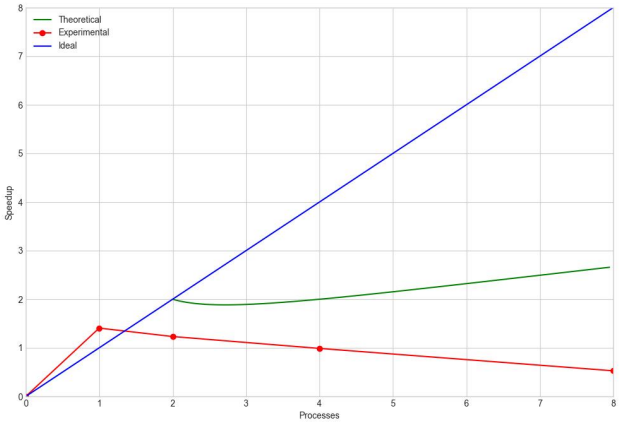

Figure 6.47: Speedup of the 2000 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.34408	0.21114	0.55522	1.0	1.0	
	Parallel	1	0.09799	0.21455	0.31254	1.77648	1.77648	
	Parallel	2	0.05011	0.38095	0.43107	1.288	0.644	
	Parallel	4	0.02663	0.4638	0.49043	1.13211	0.28303	
	Parallel	8	0.098	1.0483	1.1463	0.48436	0.06054	

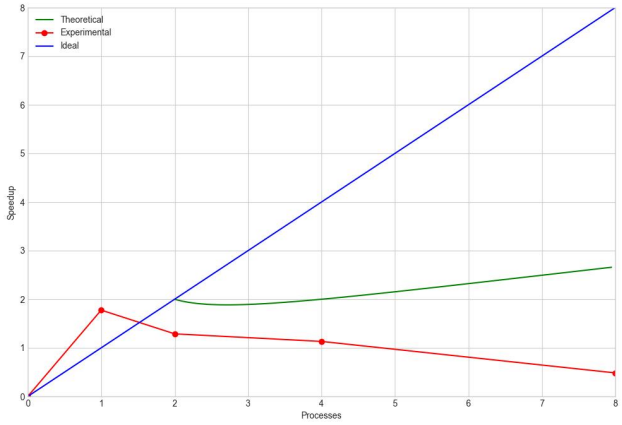

Figure 6.48: Speedup of the 2000 vertices graph with shrink 4.

6.2.6 SIZE-2400 PC

With the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.72762	1.20893	1.93655	1.0	1.0	
	Parallel	1	0.36685	1.21168	1.57853	1.22681	1.22681	
	Parallel	2	0.18249	1.35135	1.53383	1.26256	0.63128	
	Parallel	4	0.10875	1.2351	1.34385	1.44105	0.36026	
	Parallel	8	0.18285	1.94017	2.12302	0.91217	0.11402	

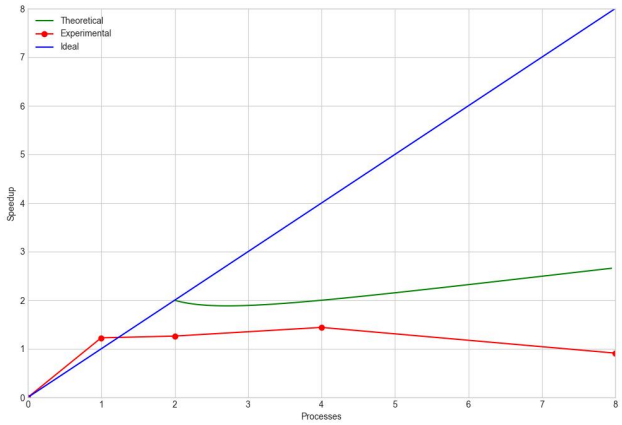

Figure 6.49: Speedup of the 2400 vertices graph with shrink 1.

With half of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.59395	0.59872	1.19267	1.0	1.0	
	Parallel	1	0.23237	0.60725	0.83962	1.42049	1.42049	
	Parallel	2	0.11701	0.84152	0.95854	1.24426	0.62213	
	Parallel	4	0.07182	0.97451	1.04633	1.13986	0.28497	
	Parallel	8	0.12704	2.2588	2.38584	0.4999	0.06249	

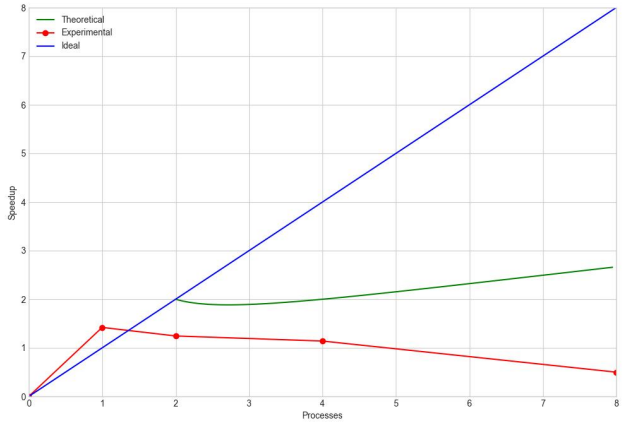

Figure 6.50: Speedup of the 2400 vertices graph with shrink 2.

With a quarter of the maximum number of edges for each vertex

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.4967	0.30599	0.80269	1.0	1.0	
	Parallel	1	0.14009	0.31237	0.45245	1.7741	1.7741	
	Parallel	2	0.07002	0.50185	0.57187	1.40362	0.70181	
	Parallel	4	0.03717	0.61845	0.65561	1.22434	0.30609	
	Parallel	8	0.10951	1.37939	1.4889	0.53912	0.06739	

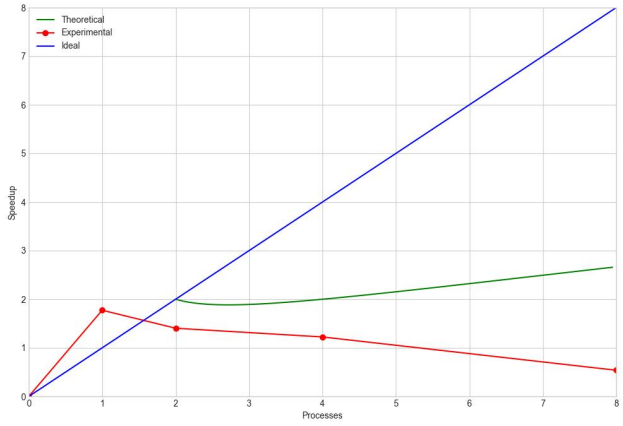

Figure 6.51: Speedup of the 2400 vertices graph with shrink 4.

6.3 Tarjan with optimization levels

Now we report the result of Tarjan's algorithm with different levels of optimization on a graph with size 2000.

6.3.1 O1

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.49726	0.5095	1.00676	1.0	1.0	
	Parallel	1	0.23695	0.51748	0.75442	1.33448	1.33448	
	Parallel	2	0.11825	0.61739	0.73564	1.36855	0.68427	
	Parallel	4	0.07	0.63672	0.70672	1.42455	0.35614	
	Parallel	8	0.13401	0.79018	0.92419	1.08934	0.13617	

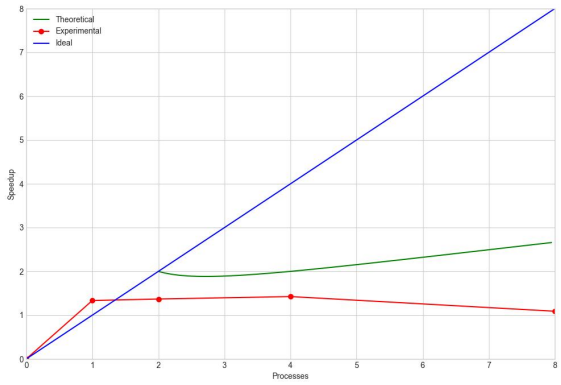

Figure 6.52: Speedup of the 2000 vertices graph with optimization 1.

6.3.2 O2

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.49382	0.50246	0.99628	1.0	1.0	
	Parallel	1	0.22914	0.5062	0.73534	1.35486	1.35486	
	Parallel	2	0.11467	0.61067	0.72535	1.37352	0.68676	
	Parallel	4	0.06333	0.52288	0.58621	1.69953	0.42488	
	Parallel	8	0.13015	0.89441	1.02456	0.9724	0.12155	

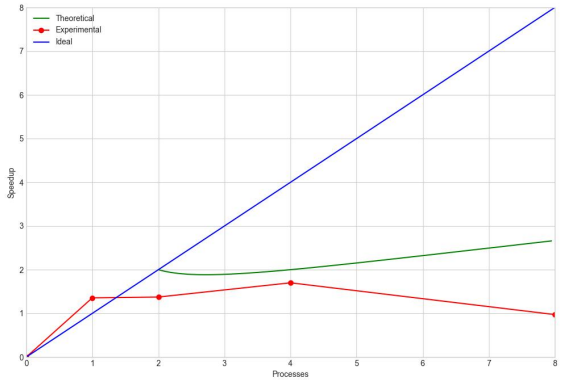

Figure 6.53: Speedup of the 2000 vertices graph with optimization 2.

6.3.3 O3

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.49294	0.50982	1.00276	1.0	1.0	
	Parallel	1	0.23028	0.50828	0.73856	1.35772	1.35772	
	Parallel	2	0.11542	0.6246	0.74002	1.35504	0.67752	
	Parallel	4	0.06147	0.56254	0.62401	1.60696	0.40174	
	Parallel	8	0.11631	0.89209	1.0084	0.99441	0.1243	

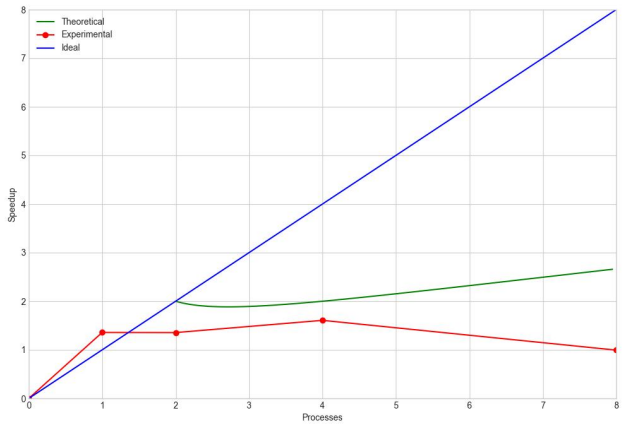

Figure 6.54: Speedup of the 2000 vertices graph with optimization 3.

6.4 Kosaraju with optimization levels

The same measures with different levels of optimization have been applied to the Kosaraju's algorithm too, on a graph of size 2000.

6.4.1 O1

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	0.49707	0.7951	1.29217	1.0	1.0
Parallel	1	0.23791	0.81582	1.05372	1.22629	1.22629
Parallel	2	0.11847	0.85884	0.9773	1.32218	0.66109
Parallel	4	0.09181	0.92415	1.01596	1.27187	0.31797
Parallel	8	0.13568	1.00375	1.13942	1.13406	0.14176

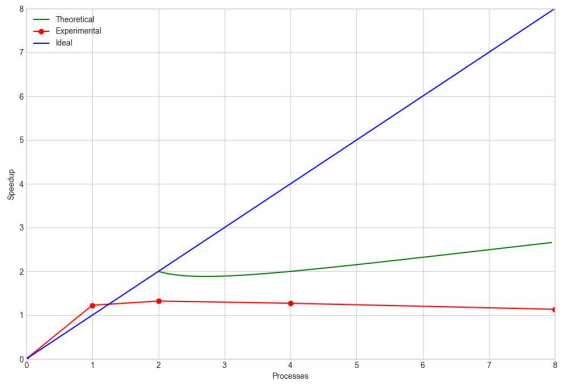

Figure 6.55: Speedup of the 2000 vertices graph with optimization 1.

6.4.2 O2

Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency
Serial	1	0.4938	0.78741	1.28121	1.0	1.0
Parallel	1	0.23277	0.78932	1.02208	1.25353	1.25353
Parallel	2	0.1145	0.84097	0.95547	1.34092	0.67046
Parallel	4	0.05953	0.7042	0.76373	1.67757	0.41939
Parallel	8	0.12933	1.26825	1.39758	0.91673	0.11459

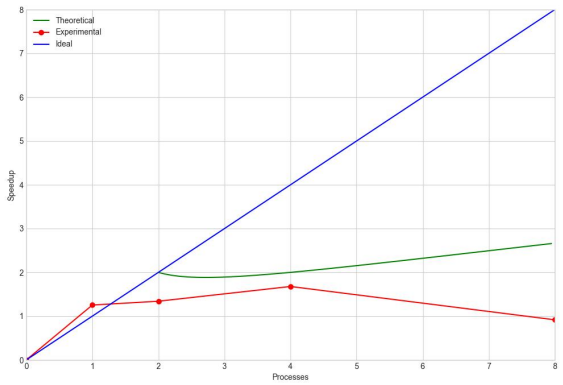

Figure 6.56: Speedup of the 2000 vertices graph with optimization 2.

6.4.3 O3

	Version	Processes	ReadFromFile	SCC	Elapsed	Speedup	Efficiency	
	Serial	1	0.49279	0.792	1.28479	1.0	1.0	
	Parallel	1	0.23108	0.79757	1.02865	1.24901	1.24901	
	Parallel	2	0.11452	0.84982	0.96434	1.3323	0.66615	
	Parallel	4	0.06556	0.75367	0.81923	1.56829	0.39207	
	Parallel	8	0.17704	1.2632	1.44024	0.89207	0.11151	

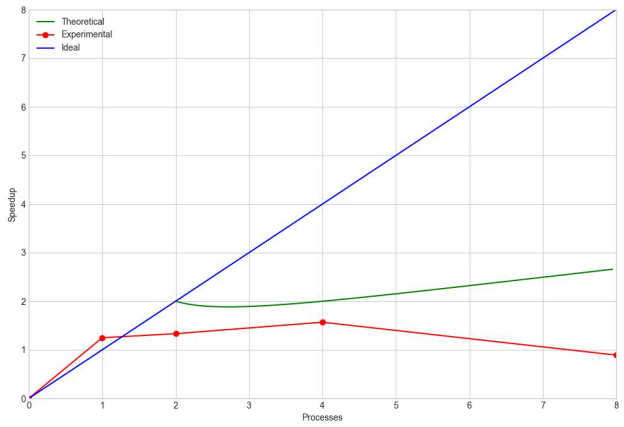

Figure 6.57: Speedup of the 2000 vertices graph with optimization 3.

Chapter 7

How to run

7.1 Single Execution

To execute the program, once all the files have been extracted in a directory, it's needed to generate the executables through the *make* command. It will manage to build all the files and update the existing ones if necessary, placing them in the *Build* folder. By using the command *'make test'* a series of tests can be executed, but it could take some time depending on the system's specifications. To manually execute the program it's possible to use the script file *run.sh*; by using *'./run.sh -h'* it's also possible to see all the options to execute the program. The output of that command will be the following:

```
Executes the program on a given number of processes. If the
  [run serial] parameter is not inserted
it will execute it as default

Usage:  ./run.sh [num of processes]
        ./run.sh [num of processes] [run serial (0/1)]
        ./run.sh -h     -to print this help output
        ./run.sh -t     -to execute a series of tests
```

Listing 7.1: The output of "*./run.sh -h*"

To further customize the program's execution it's needed to modify the file *Headers/Constants.h* with the desired values; once done it's needed to rebuild the whole program by executing *'make -B'*. It's also possible to customize the compilation of the program, specifying compilation options and more to the *make* command. Using 'make help' a list of all the available options will be printed on the screen; The output of the command will be the following:

```
Builds all the binaries required to test the program
Usage:  make all       -the default way to execute make,
                          builds all the required files
        make test      -runs various tests of the program
        make clean     -cleans the working directory,
                          deleting object files and eventual output files
        make cleanObj  -cleans the working directory, deleting
                          the object files
        make cleanBin  -cleans the working directory, deleting
                          the binary output files
        make cleanTxt  -cleans the working directory, deleting
                          the textual output files
        make optimizeX -builds all the required files applying
                          the specified level of optimization X=[1-3]
        make help      -prints this output
```

Listing 7.2: The output of "*make help*"

To get various measures and graphs of the obtained speedups there are a few steps to do:

1. Run *'./measures.sh'* to execute a series of tests
2. Run *'pipenv install'* to set up the virtual environment to generate the graphs
3. Execute *'pipenv run python Measures/extract.py'* to analyze the results and generate the graphs

7.2 Test Cases

In order to make some tests both on the Tarjan's and Kosaraju's algorithms, it is possible to just execute *make test*. The *Test.sh* script will be executed by this command. The script executes another script, called *Run.sh*, which executes both the parallel and the serial version of both the algorithms, comparing them and printing *"Completed successfully"* in case they all returned the same result, an error message otherwise. Different possible configurations are tested: changing the number of processes, the number of edges for each vertex, and the size of the graph.

```
ARRAY_SZ=(400 800 1200 1600 2000 2400)
ARRAY_PROC=(1 2 4 8)
ARRAY_SHRINK=(1 2 4)

for graphSize in "${ARRAY_SZ[@]}"; do
    for proc in "${ARRAY_PROC[@]}"; do
        for shrink in "${ARRAY_SHRINK[@]}"; do
            echo -e "\t"Testing executed on a $graphSize vertices graph
            echo -e "\t"Testing executed with $proc processes
            echo -e "\t"Testing executed with $shrink shrink factor
            wl=$(($graphSize/$proc))
            min=$(($graphSize/$shrink))
            min=$min-10
            max=$(($graphSize/$shrink))
            ./Source/updateConstants.sh $wl $min $max
            make -B
            ./run.sh $proc
        done
    done
done
```

Listing 7.3: The test code that executes different testcases

Chapter 8

Conclusions

Taking into account the results of the testing phase of both the algorithms, it's noticeable how the parallel version is almost always faster than the serial one with a great number of edges for each vertex. Despite this, when the program is executed with a number of processes that is greater than the number of physical processors of the device, or if the number of processes is too big compared to the size of the graph, the improvements of the parallel version decrease.

8.1 Measures on the Cluster

Let's analyze the results obtained executing Tarjan's algorithm on the cluster. In general (and this kind of behavior is shown in the measurements made with the PC too) the improvements in terms of speedup increasing the number of processes working on the algorithm are better shown in the case with a huge amount of edges for each vertex of the graph (which means longer adjacency lists and fewer macro nodes resulting from the algorithm at each iteration). In contrast, as long as the number of edges for each vertex decreases, the time spent in the communication makes the SCCs computation much more slowly than the serial version (which has not got the overhead of the communication). Also, the size of the graph plays a relevant role in affecting the speedup since it changes the workload of each process and so the amount of work that each process has to do to read the SubGraph from its own file, convert it into a ListGraph, apply the algorithm for the SCCs computation, communicate the subgraph and the result, ... The best performances are obtained when each subgraph is small enough to allow every single process to compute the SCCs fast but at the same time big enough to consider the overhead of the communication irrelevant despite to the advantages of the parallelization. Since the front-end node has worse performances than the back-end ones, the obtained speedup has to be carefully interpreted: we can reach a huge speedup value (e.g. $\sim$ 6 with 16 processes in the graph with 800 or 1200 nodes), but this is mostly due to the difference in terms of performances among the nodes, which allows compensating, and sometimes surpass, the overhead introduced by MPI. Despite this, we can still extract relevant information from these graphs, such as the general trend of the graph: with a lot of edges for each vertex and the right amount of workload for each process, performances increase with the number of processes, otherwise, they generally decrease in speedup when the number of processes increases.

8.2 Measures on the PC

Now we consider the measures made on Tarjan's algorithm with the PC. Before we start analyzing the result, we want to underline that the number of cores of the processor is 4, so we do not expect better performances increasing the number of processes to more than 4. The maximum amount of speedup we managed to obtain in this case is 1.5 $\sim$ 2 in correspondence of 4 processes (for the reason reported above). The general trend shown in these graphs is the same

as the ones of the cluster: with a great number of edges for each vertex performances improve with the number of processes, otherwise, they decrease, this is true as long as there's a balance in the workload of each process. The measures made on Kosaraju's algorithm show almost the same behavior as Tarjan's one. We imagined that this could happen since the whole parallel execution is the same, except for the SCCs' computation which performs an additional DFS both in the serial and in the parallel version. Other additional measures are contained in the *Measures* folder of the project, they have not been reported here because they present similar behaviors.

8.2.1 Cluster vs PC measures

It is interesting to focus for a second on the differences in terms of elapsed time of the measures made on the cluster and the ones made on the PC. The second ones, as a matter of fact, are significantly lower than the first ones. This is, of course, due to the differences in terms of hardware, but also to the fact that some of the processes had to communicate from different devices, so some further latency was introduced. Despite this, the overall behavior is the expected one in proportion and taken into account all the aspects mentioned before.

8.3 General findings

Generally, it's possible to find a kind of correlation between the speedup and various factors:

- **The size of the graph**. This factor affects the workload of each process and consequently the maximum amount of edges for each vertex.
- **The number of edges of the graph**. The parallel algorithm runs faster than the serial on dense graphs since it allows the algorithm to find more SCCs in the first iterations, decreasing the computation of the next ones.
- **The optimization level of the compiler**. The compiler's optimization helps the program have smaller times without interfering with the MPI platform. It was noticed that the best optimization level was the second one.

8.4 Further considerations

By better analyzing the read and SCC times of the tests it's also possible to see an interesting behavior: while the overall time for the parallel execution is less than the one required for the serial one, the time required to execute just the SCC algorithm is actually a lot worse in some cases. By considering just the SCC algorithm, in such cases, the speedup would have been equal to about 0.05, against the $1.5 \sim 2$ of the whole execution. Is also to be considered that the serial program reads the file using the *fscanf* function and reads *integers*, while the parallel one uses the MPI primitives to read binary files, resulting in single process parallel times which are twice greater than the serial ones. In the majority of cases, it is the read from file part which assures us a significant speedup in the parallel version. This explains the strange behavior of the graphs with a small amount of edges for each vertex, which report a speedup of almost 2 in correspondence of the execution with just one process in MPI, we now know that this is due to the fact that the one using MPI takes half the time of the serial one to read the graph. This means that the real experienced speedup is only due to the file access approach, which was a likely result, knowing that algorithms working on graphs, given their general recursive nature, are rarely parallelized, given to the difficulty to exploit the benefit of parallel computation.

8.5 License