University of Salerno

Department of Information and Electrical Engineering and Applied Mathematics

Master Degree in Computer Science

High Performance Computing

A.A. 2024/2025

Professor:
Francesco Moscato

Student:
Lorenzo Pepe
matr.: 0622702121

Contents

Introduction

This report summaries the work done for the examination of "High Performance Computing and Big Data", held by the professors Moscato Francesco and D'Aniello Giuseppe, for the master's degree course in Computer Science of the Department of Information and Electrical Engineering and Applied Mathematics, in the academic year 2024/2025.
This report cover only the HPC part of the project, that is the parallelization of the VF2++ algorithm, with the MPI technology.

The report covering the the Big Data project has being delivered separately.

1. Introduction to VF2++ and the parallelization technologies

In this chapter the *VF2++* algorithm will be briefly introduced, alongside with the three parallelization technologies presented in the course, that are:

- ***MPI***: Message Passing Interface;
- ***OpenMP***: Open Multi-Processing;
- ***CUDA***: a proprietary parallel computing platform and API based on GPU processing power and architecture, developed by NVIDIA.

1.1 VF2++ algorithm

VF2++ is the newer implementation of the *VF2* algorithm, faster and than the previous one. *VF2* is an algorithm used to find isomorphism into graphs; an isomorphism, in mathematics, is a structure-preserving mapping between two structures of the same type that can be reversed by an inverse mapping. More specifically, in graph theory, an isomorphism between two graphs G and H is a bijective map f from the vertices of G to the vertices of H that preserves the "edge structure" in the sense that there is an edge from vertex u to vertex v in G if and only if there is an edge from *f(u)* to *f(v)* in *H*. In simple words, two graph are isomorph if they have the same number of nodes, with the same connection, but organized in two distinct forms. The images below show the concept just formalized.

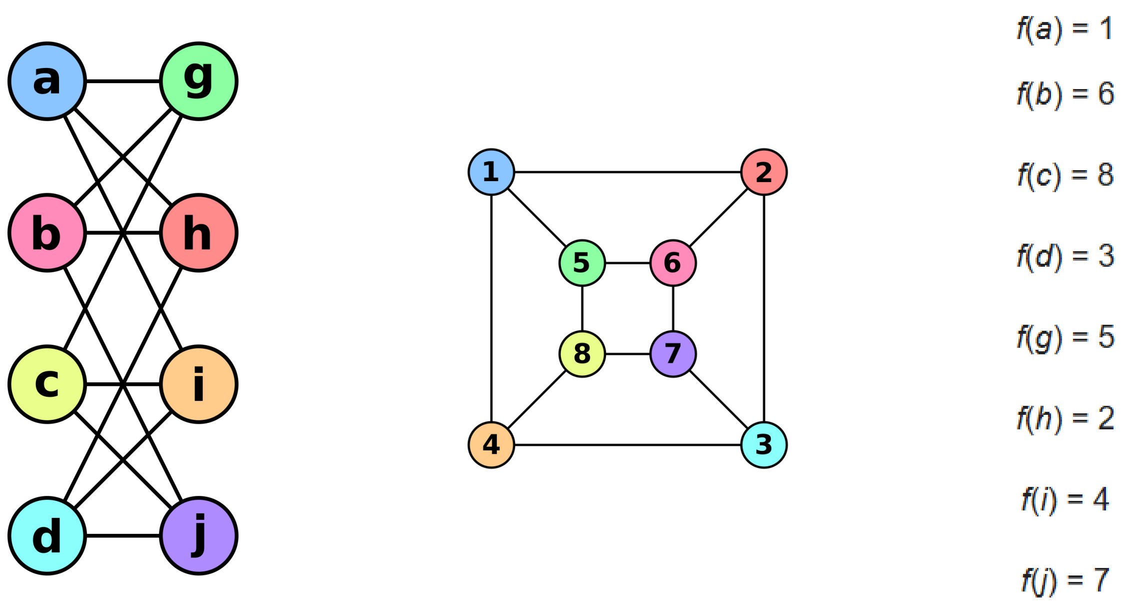


Figure 1: Two isomorph graphs and their relation [1]

The differences between the two algorithm versions, as described in Jüttner et al.[2], are basically two:

- firstly, taking into account the structure and the node labeling of the graph, *VF2++* determines a matching order in which most of the unfruitful branches of the search space can be pruned immediately;
- secondly, introducing more efficient—nevertheless still easier to compute — cutting rules reduces the chance of going astray even further.

All other information about the algorithm can be found in the above-mentioned work, along with the mathematical formal definition.

1.2 Parallelization technologies

Parallelization is the process of dividing the workload into smaller and easier sub-tasks, executed simultaneously, rather than sequentially. This can be done using a single processors, exploiting all it's cores at once, rather than using only one (therefore using the real capacities of the processors), or sharing the workload between more processors at once. The goal of parallelization is to improve the efficiency and speed of computations.
Parallelization involves the use of the "*divide et impera*" paradigm before delving into the actual code: it is vital to know how to parallelize an algorithm before doing it, avoiding the try and error approach.

1.2.1 Message Passing Protocol

The *Message Passing Interface* (MPI) is a standardized means of exchanging messages between multiple computers running a parallel program across distributed memory. It defines useful syntax for routines and libraries in programming languages including Fortran, C, C++[3].

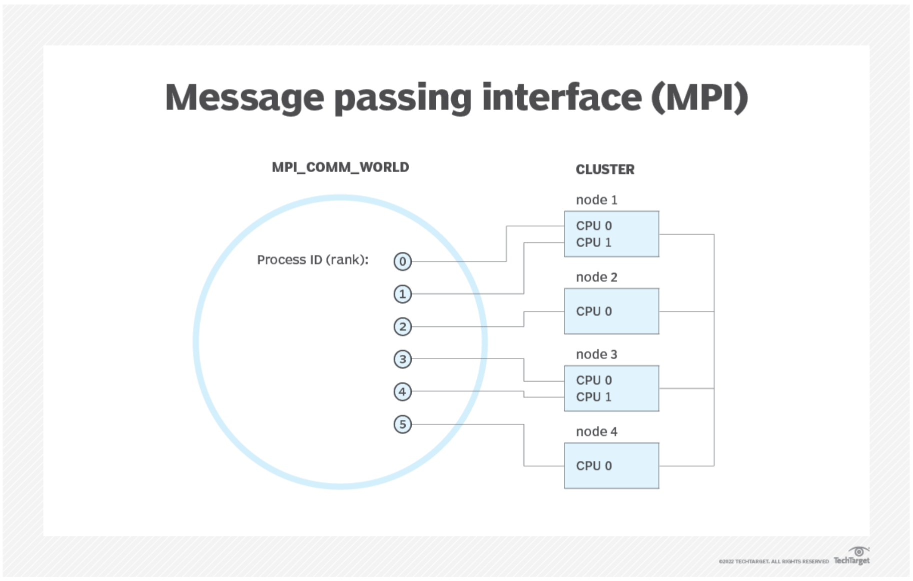


Figure 2: An MPI process containing multiple nodes in four clusters shows how a rank is given to each CPU[3]

MPI has several benefit, other than the obvious advantage of cutting down computational time in softwares:

- ***standardization***: since MPI has replaced other message passing libraries, it became a generally accepted standard for parallelization;
- ***portability***: MPI has been implemented for many distributed memory architectures, meaning users don't need to modify source code when porting applications over to different platforms that are supported by the MPI standard;
- ***functionality***: MPI is designed for high performance on massively parallel machines and clusters.

In addition, despite may not be an official standard, MPI is still a general standard created by a committee of vendors, implementors and users.

1.2.2 Open Multi-Processing

OpenMP is the second technology introduced in the HPC course. It's an implementation of multi-threading, a method of parallelizing whereby a primary thread (a series of instructions executed consecutively) forks a specified number of sub-threads and the system divides a task among them. The threads then run concurrently, with the runtime environment allocating threads to different processors.
It is based on markers, namely special part of the code that "marks" code zones that have to be executed in parallel way, so the compiler can create the necessary threads before the execution. Each thread has an ID and the main thread have always the number zero. After the execution of the parallelized code, the threads join back into the primary thread, which continues onward to the end of the program.
The image shows the execution of a program, where the main line represents the main thread, that splits into several separate thread, executing code in a parallel way.

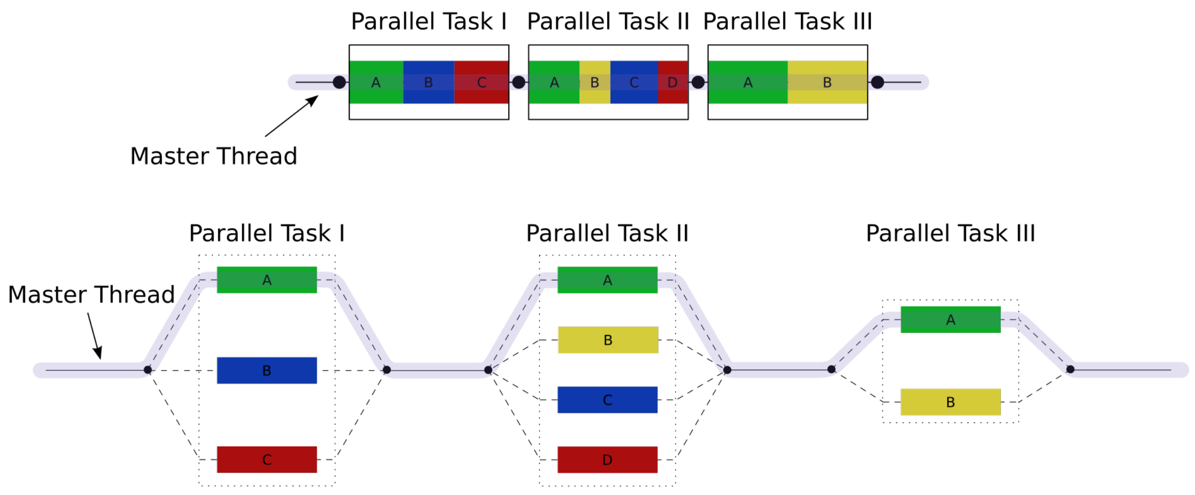


Figure 3: OpenMP behaviour[4]

Advantages of using OpenMP are:

- it is easy to use and learn since parallelize existing code can be done by adding a few directives or *pragmas* that tell the compiler how to distribute the work among the available threads;
- the number of threads can be decided a priori, such as the scheduling policy, the data sharing and the synchronization mechanisms with simple clauses and runtime functions;
- it is also portable and widely supported by most compilers and platforms, and so can run your code on different systems without much modification.

OpenMP is designed for shared-memory systems, where all the threads can access the same memory space. This introduces some challenges:

- **scalability**: since all the threads must access the same memory areas, this implies that the memory bandwidth and the cache coherence can become bottlenecks as the number of threads grows;
- **performances portability**: OpenMP does not guarantee that the same code will run equally fast on different systems, as it depends on the compiler optimizations, the hardware features, and the runtime environment.

To use OpenMP in a safe way, some best practices and guidelines were created to which all must adhere to maintain the code as mush clean and readable to overcome the aforementioned challenges, which are: using the latest version of the OpenMP standard, use the appropriate level of parallelism (balancing the workload among threads), minimizing data sharing and synchronization among threads is also essential, as is optimizing memory access patterns and cache usage.

1.2.3 CUDA

Created by Nvidia, CUDA is a proprietary parallel computing platform and API that allows software to use certain types of graphics processing units (GPUs) for accelerated general-purpose processing. It is a software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements. CUDA is designed to work with programming languages such as C, C++, Fortran and Python. The main advantage of CUDA is that it allows for parallel processing, harnessing the power of GPU cores, which can handle thousands of threads simultaneously, offering a significant speed advantage over traditional CPU processing for certain tasks. It is also well-documented and supported, with many tools and libraries for deep learning and for parallel algorithms. CUDA scales well across different generations of NVIDIA GPUs, from consumer-grade to high-end data center. CUDA has a wide range of applications going from machine learning model training and inference to high-performance scientific simulations, image and video processing, cryptographic computations, game development and even financial modeling and risk analysis.

However, CUDA has some drawbacks:

- CUDA works exclusively with NVIDIA GPUs, which ties developers and organizations to NVIDIA's ecosystem;
- despite extensive documentation, CUDA programming requires knowledge of GPU architecture and optimizing code for GPUs (e.g., minimizing memory transfers or avoiding warp divergence) can be challenging;
- debugging CUDA applications can be more complicated compared to traditional CPU applications, especially when dealing with synchronization issues or memory access errors;
- high-performance NVIDIA GPUs, such as the A100 or H100, are expensive and may not be affordable for small-scale projects.

2. System specifications and technologies utilized

In this chapters the technologies used will be introduced and justified, along with the specifications of the system used for testing and the MPI libraries setup.
The hardware used for the project is a personal computer, in the specific an *HP Omen Laptop Series 16-c0*, consisting of the following components:

- **CPU**: AMD Ryzen 7 5800H, with the following specific:

Socket	AMD Socket FP6
Foundry	TSMC
Process size	7nm
Transistors	10,700 million
Die size	180 mm^2
Package	FP6
tJMax	105° C
Frequency	3.2 GHz
Turbo Boost	up to 4.4 GHz
Base Clock	100 MHz
TDP	45 W
Generation	Ryzen 7
Memory support	DDR4
Rated speed	4266 MT/s
Memory bus	Dual-channel
ECC Memory	No
PCI-Express	Gen 3
Cores	8
Threads	16
SMP # CPUs	1
Integrated Graphics	Radeon Vega 8
Cache L1	64 KB (per core)
Cache L2	512 KB (per core)
Cache L3	16 MB (shared)
Features	MMX, SSE, SSE2, SSE3, SSSE3, SSE4A, SSE4.1, SSE4.2, AES, AVX, AVX2, BMI1, BMI2, SHA, F16C, FMA3, AMD64, EVP, AMD-V, SMAP, SMEP, SMT, Precision Boost 2.

- **RAM**: 16 GB, of which 15.3 GB useable;
- **GPU**: AMD Radeon 6600M (altough this will not be used since the project requires MPI and not CUDA, which is not supported by AMD);
- **System**: Ubuntu 24.10 .

2.1 Software development

The full *VF2++* software development and test follow few steps:

- simple graph software development: the first step is to implement a base graph building algorithm, based on adjacency lists; the choice for this specific data structure finds support in the complexities that the adjacency lists offers, discussed shortly;
- addition of graph construction from files: this is needed for simplicity's sake, since the graphs to be loaded in memory have at least 10 thousands nodes and various coverage percentage (from 30% to 70%);
- implementation of the *VF2++* serial features and execution times testing;
- use of MPI for parallelization of VF2++ and testing of the new execution time;
- additionally, use of MPI for parallelizing the graph creation, with additional tests of the execution times;
- comparison between all the execution times, with different optimization options.

The choice of the adjacency list data structure is simple: compared to the others choices, such as adjacency matrix, it's faster. Indeed, in a graph G with V vertices and E edges:

- the **Space Complexity** is *O(V+E)*;
- the **Time Complexity** for checking if an edge from the vertex v to the vertex w exists is *O(degree(v))*;
- the **Time Complexity** for the insert operation is *O(1)*.

3. Graph and VF2++ serial implementation

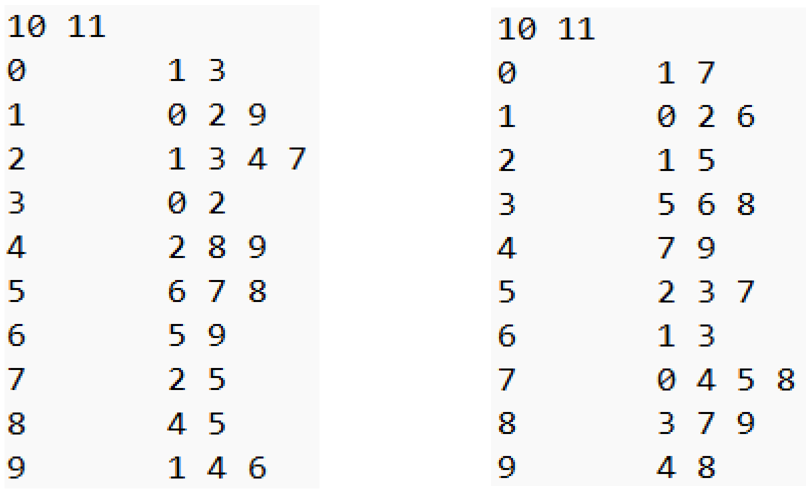
In this chapter, the graphs and the VF2++ serial code will be explained, with timing tables reporting the time taken by the serial code for various types of graph.

3.1 Graph implementation

As already stated in the last chapter, the graph structure uses an adjacency list to store the neighbors of each node. The structures themselves are simple, being just a Node and a Graph struct.
In particular, the file used for storing the graphs, follow a specific pattern:

- the first line consists of two numbers representing the total number of nodes of the graph and the total lines present in the text file, separated by a space character;
- from the second line on, each line is formatted with at least one number, representing the ID number of the node, a tab character (\t) and a list of number that constitute the adjacency list of that specific node; each number in the list is separated by a space character.

An example of two isomorphic graphs, with 10 nodes and a coverage of 30%, in a text file is provided in the figure below.



```
10 11
0       1 3
1       0 2 9
2       1 3 4 7
3       0 2
4       2 8 9
5       6 7 8
6       5 9
7       2 5
8       4 5
9       1 4 6
```

```
10 11
0       1 7
1       0 2 6
2       1 5
3       5 6 8
4       7 9
5       2 3 7
6       1 3
7       0 4 5 8
8       3 7 9
9       4 8
```

Figure 4: Token (left) and Pattern (right) isomorphic graph

The graph generation is provided by a Python program, that generate first a graph with the exact characteristic required by the user (given via command line) and then a specific number of isomorphic graph (also this value is passed via command line) with the same number of nodes and same coverage, using random connection for creating the first graph and permutations to generate all the other graphs, isomorph to the first one.

The algorithm for loading the graph in memory follow simple step: the file is opened and parsed, reading the first line, which contains the number of nodes. The a graph is created, by allocating only the nodes. Then for each node, it's adjacency list is loaded, until the file has been parsed completely. Note that the graph creation supports node without edges, so there can be nodes isolated in a graph, in this implementation.
Lastly, the algorithm manages several error handling, such:

- file path missing;
- file not opened correctly;
- badly parsed files;
- other errors caused by bad conversions when parsing the nodes lines.

3.2 VF2++ Implementations

VF2++ is the new implementations of VF2, an algorithm for searching isomorphism and sub-isomorphism in graphs. It has various application, from networking to DNA pattern search. The new version implements two major update to the algorithm:

- a different approach to the matching order problem;
- cutting techniques that prune those part of the graph that are useless for the searching part.

Together these two new techniques reduces the search space, and the time, considerably. The complete code for the VF2 algorithm is available in the GitHub page of the MiviaLab[5], the research Lab of the University of Salerno active in the fields of Pattern Recognition and Computer Vision, while the VF2++ algorithm is described in article published by Alpár Jüttner and Péter Madarasi.[2].

The C implementations compute the matching order before cited, based on the degree of each node. We may not distinguish between outdegree and indegree for the graphs examined in this work, since only undirected graphs are used, and use only degree to indicate all the edges connected to a node. The algorithm search all the nodes and it's degree where, nodes with a large coverage have a stricter constraints to satisfy. Generally, we want to match nodes based on their degree, so nodes with a high degree have less probability to be matched, while nodes with a low degree can be mapped to more nodes in the other graphs.
The algorithm select the nodes with the highest degree in the Token graph, and use it has a starting node for the BFS (Breadth First Search) traversal algorithm. On each layer of the BFS, nodes are sorted based on their degree in a descent way. Doing so we are sure to analyze first the node with higher degree, making the search faster in the next BFS layers. The BFS search ends when all the nodes in the G1 graph has been analyzed and sorted.

Next, the algorithm uses the G1's nodes sorted to find the candidate for isomorphism in G2. Each G1's node is confronted with G2's node. In particular, if we define:

$$(u, v) \text{ is a pair of node } | \; u \in G_1, v \in G_2$$

then the confrontation is computed as follow:

- if u has no already mapped neighbors, all the nodes in G2 are possible candidates, provided they meet the following conditions:
  - have the same number of neighbors of u, thus the same degree;
  - are not already in the mapping;
  - have no neighbors already mapped;
- if u has already one neighbor mapped, candidates are the neighbors of this mapped node that:
  - have the same number of neighbors as u;
  - are not already included in the mapping;
- if u has more than one neighbor already mapped, the candidates are the nodes in G2 that meets all the following conditions:
  - they belong to the intersection of the mapped neighbors of u;
  - have the same number of neighbors as u;
  - they are not already included in the mapping.

If a candidate for u has been identified, the node and it's list of candidates are saved on the stack, where is the mapped to the first available candidate. In the case that there are no candidates for the node u, the algorithm executes backtracking, examining the last node in the stack, mapping it to a next candidate among all those still available.

If the stack is completely depleted without finding a full mapping for all the nodes in G1, it means that the two graphs are not isomorph. If all the nodes in G1 are mapped to the nodes of G2, it means that the two graph are indeed isomorph.

In the end, the computational time is acquired using the "$clock()$" function, to get the number of clock ticks elapsed since the program was launched. The start time is acquired just before the creation of the graph paths, while the end time is obtained after the two free functions. The computational time, in seconds, is obtained with the equaiton:

$$cTime = \frac{endTime - startTime}{CLOCK\_PER\_SEC} \tag{3.1}$$

All the performances, and the modus operandi for acquiring the computational times, are discussed in Chapter 5, after the explanation of the parallel algorithm version, in such a way that we can compare the serial and parallel execution times.

4. Use of MPI for VF2++ parallelization

The parallel program for VF2++, using MPI, inherit all the characteristic of the serial version. In fact, all the parallelization is done in the main file, so the workflow for the VF2++ is the same as it's serial version, with the difference that each process load it's own graph and, after it have received the main graph, execute the algorithm to check for eventual isomorphism.
To be able to send the main graph to all the workers, a "Flat Graph" has been implemented, which is described in the next section.

4.1 Flat graph implementation

Sending the graph structure via MPI broadcast is not possible, since the graph itself is a pointer. In order to solve this problem, a "Flat graph" logic has been implemented.
A Flat Graph is a 1D array of integers, which serializes the hierarchical graph structure (nodes and their adjacency lists) into a linear format that can be broadcast over MPI. It has three main advantages:

- **Compactness**: uses contiguous memory for storing all the graph data, compliant with both MPI Broadcast and Send function;
- **Self-descriptive**: the array start with the total number of nodes; for each node is then present, in a contiguous way, all of it's data (such as ID, mapped status, number of neighbors and it's entire neighbor list), so parsing the array is simple;
- **Variable in size**: since the array is based on an already existing graph, it's size can be computed beforehand and therefore a contiguous memory area can be assigned using specific functions (such as malloc).

In addition the serialization and deserialization is executed only on the main graph, so these method can efficiently save space, since each worker load it's own graph in the typical structure without serializing/deserializing each graph. The function that execute the serialization, takes the standard graph and compute the size of the array.
Final, since this is a 1D array, it's possible to use the standard $free()$ function already present in the C *stdlib* library.

In this version, the program read a file which contains all the filenames of the graphs it needs to load: these names are not passed manually as was for the serial version of the program. It then recreates the full paths for every graph in the list, taking note that the first filename in the list is the name of the main graph.

4.2 Main workflow

The workflow in the main file is divided into three major parts:

- **Start/End part**: in this part there is all the code responsible for parsing the command received from the console and the function to initialize/finalize MPI;
- **Rank 0**: Rank 0 is the Master process that have a variety of tasks, which are in order of execution:
  - 1. read the main file, containing all the filenames;
  - 2. distributes the filenames among all the MPI workers, utilizing a round-robin method;
  - 3. compute the full main graph path and load it in memory;
  - 4. flatten the main graph;
  - 5. send the size and the actual flattened main graph to all the MPI workers;
  - 6. wait, collect and print all the results coming from the workers;
  - 7. free up the memory used for the flattened graph and the main graph.
- **Every other Rank**: every other rank that is not zero, is a Slave process, meaning that it has to execute a number of tasks, like the Master rank, but it has first to receive the data from the Master Rank. Each Slave process executes the following tasks, in order:
  - 1. receive the filenames assigned and compute the full path for it;
  - 2. load it's assigned graph;
  - 3. receive the size of the flattened graph and the flattened graph itself;
  - 4. recreate the original main graph, from the received flattened graph;
  - 5. launch the VF2++ algorithm and check for isomorphism between the reconstructed main graph and the graph that has been assigned to it;
  - 6. send the result to the Master rank and free the memory used for it's graph and the received flattened graph.

Then, the code has been further modified to be able to get the computation time for the entire program. Specifically the Master Rank measure the computational time for the the entire run, using the "$MPI\_Wtime()$" function. The time recording start right after the "$MPI\_Init() function$" and finish after the "$free()$" and "$freeGraph()$ functions.
In the next chapter the modus operandi for acquiring the right computational time will be explained, before showing the relevant tables and graphs.

5. Results

All the computational times has been gathered as a mean of twenty consecutive runs of the same program, with the same graphs, both in the serial and parallel version. Each run gather it's times and saved them in a .CSV file, reporting also the optimization used. The file is saved with a specific name, which specify the optimization used, how many workers has been used (if is the parallel version), the number of nodes, the coverage percentage. In order to facilitate the execution, a bash script is included in the directory which can be used to execute a number between 2 and 20 runs, with different optimizations (default is -O2) and several other parameters. The README files in each folder can further clarify each parameters, compulsory or optional, that the script can accept.

The tests executed cover four type of optimization, which are -O0, -O1, -O2 and -O3. Each optimization has been used to test graph with 500 and 1000 nodes, with coverage percentages of 30, 60 and 90. The serial version, compare two graphs, while the parallel version use 2, 4 and 8 worker, in order to be able to compare, respectively, 2, 4 and 8 graphs. More precisely the parallel program compare a specific main graph with "$n-1$" other graphs, with n being the total number of workers; so, for example, if we use 8 workers:

- Rank 0 load the main graph and pass it to the remaining 7 workers;
- all the other Ranks, receive a graph to load and the main graph, and compare the main graph with the one they load in memory.

So with 8 workers the program execute only seven comparisons. This is in order to avoid giving too much work to the Master Rank, which is responsible of organizing and sharing all the work, including paths and the main graph, via broadcast functions. Additionally the master rank must listen for incoming results from the all the other workers. All of the gathered computational times, are then used to compute the speedup using the following equation:

$$S = \frac{T_s}{T_p}, \tag{5.1}$$

where:

- S is the speedup computed;
- T_s is the execution time computed on the serial program;
- T_p is the execution time computed on the parallel program.

Another performance measure is the efficiency, computed as:

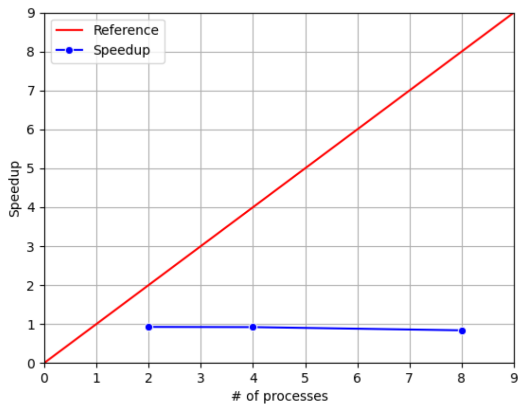
$$E = \frac{T_s}{P * T_p}, \tag{5.2}$$

where:

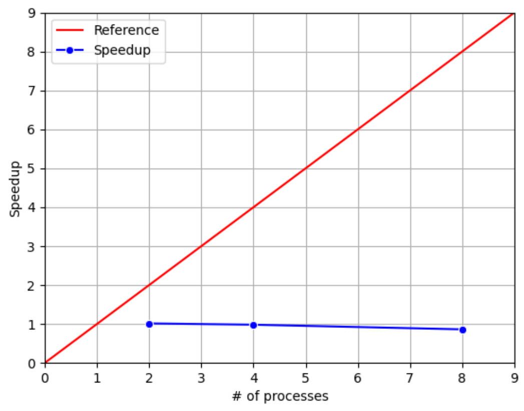
- E is the efficiency value computed;
- P is the number of processes.

5.1 Optimization: -O0

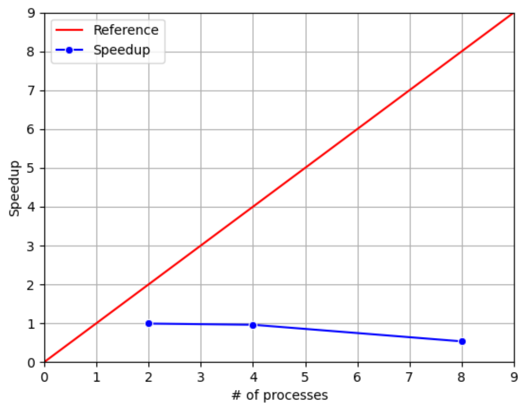
The -O0 optimization is the first level of optimization. It roughly means "no optimization" at all, meaning that the C compiler will nearly not optimize the code so that the executables are larger and slower, but with a faster compilation time.



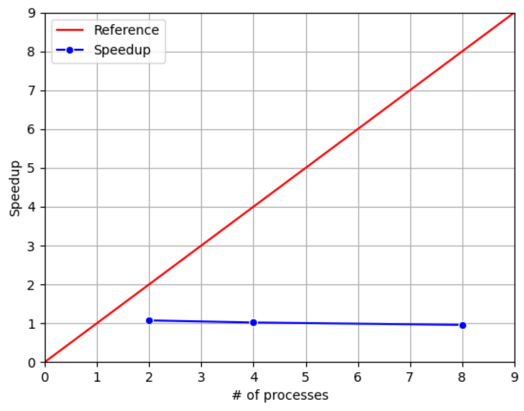
(a) 500 nodes and 30% coverage



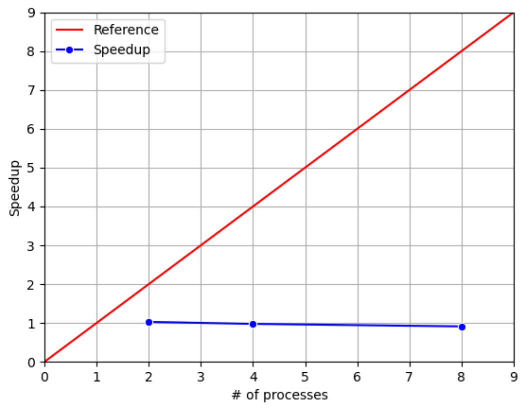
(b) 500 nodes and 60% coverage



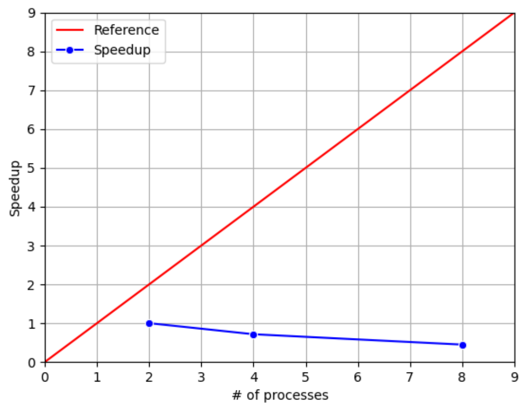
(c) 500 nodes and 90% coverage



(d) 1000 nodes and 30% coverage



(e) 1000 nodes and 60% coverage



(f) 1000 nodes and 90% coverage

Figure 5: Speedup for Optimization -O0

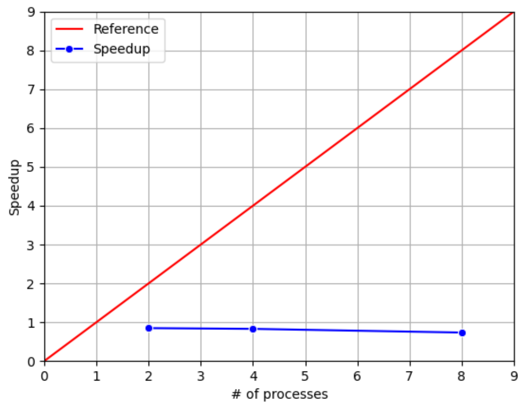
In the table below are reported all the numerical data for this optimization.

Nodes	**Coverage**	**Configuration**	**Mean exec time (s)**	**Speedup**	**Efficiency**
500	30%	Serial	0.014335	1.00	1.00
		Parallel -n 2	0.01549	0.9254	0.4627
		Parallel -n 4	0.015575	0.9204	0.2301
		Parallel -n 8	0.0171	0.8383	0.1048
	60%	Serial	0.03503	1.00	1.00
		Parallel -n 2	0.034505	1.0152	0.5076
		Parallel -n 4	0.035685	0.98154	0.2454
		Parallel -n 8	0.04059	0.8630	0.1079
	90%	Serial	0.1627	1.00	1.00
		Parallel -n 2	0.164095	0.9912	0.4956
		Parallel -n 4	0.1694	0.9602	0.2400
		Parallel -n 8	0.30496	0.5333	0.0667
1000	30%	Serial	0.070005	1.00	1.00
		Parallel -n 2	0.06528	1.0724	0.5362
		Parallel -n 4	0.06868	1.01929	0.2548
		Parallel -n 8	0.073185	0.9565	0.1196
	60%	Serial	0.19827	1.00	1.00
		Parallel -n 2	0.1927	1.0289	0.5145
		Parallel -n 4	0.202985	0.9768	0.2442
		Parallel -n 8	0.217325	0.9123	0.1140
	90%	Serial	0.5555	1.00	1.00
		Parallel -n 2	0.554755	1.0013	0.5007
		Parallel -n 4	0.775455	0.7163	0.1791
		Parallel -n 8	1.2364	0.4493	0.0562

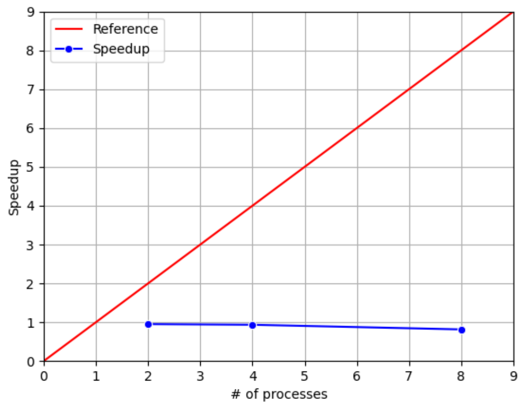
Table 1: Mean execution time, Speedup and Efficiency for OPT -O0

5.2 Optimization: -O1

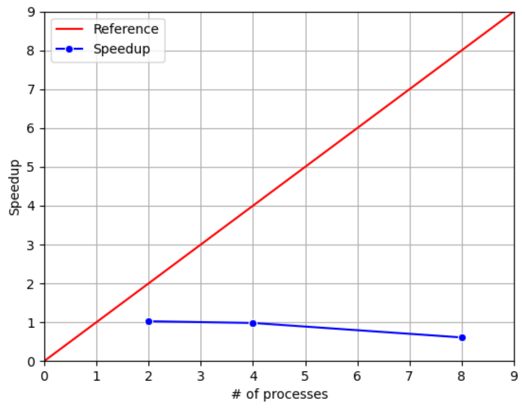
This is the second level of optimization. At this level the C compiler executes a number of optimization to improve performance without significantly increasing compilation time or executables file size.



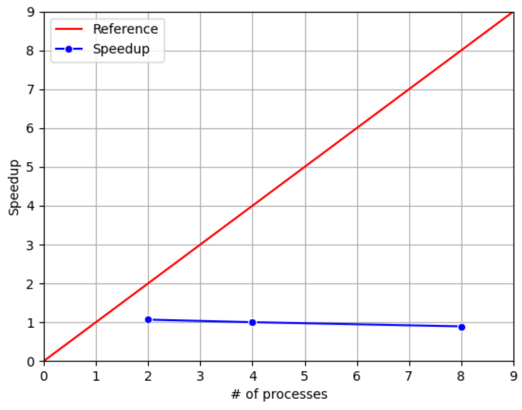
(a) 500 nodes and 30% coverage



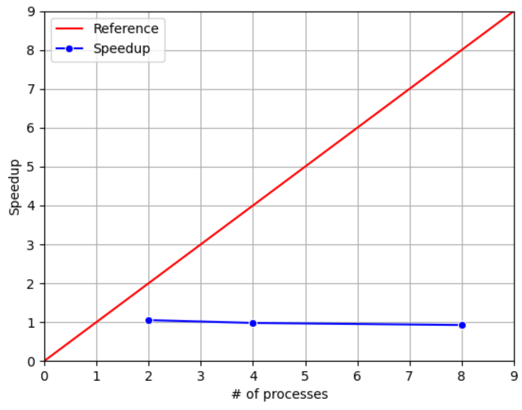
(b) 500 nodes and 60% coverage



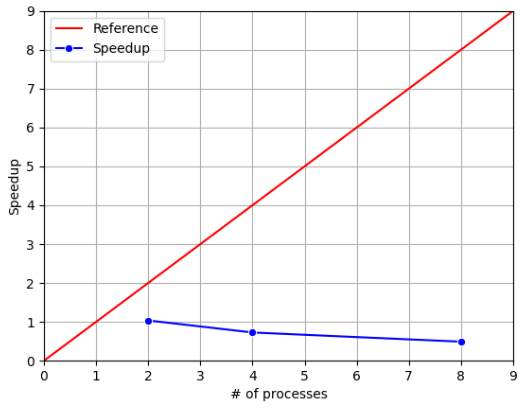
(c) 500 nodes and 90% coverage



(d) 1000 nodes and 30% coverage



(e) 1000 nodes and 60% coverage



(f) 1000 nodes and 90% coverage

Figure 6: Speedup for Optimization -O1

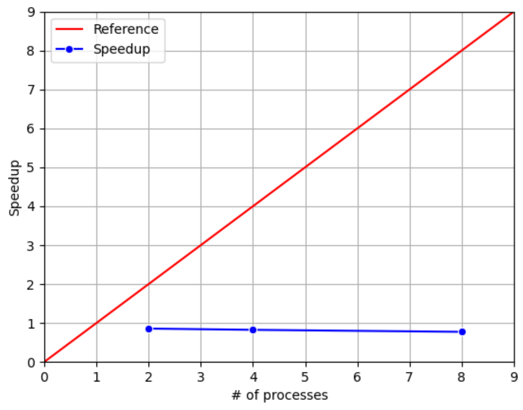
All numerical data for this optimization are reported in the table below.

Nodes	**Coverage**	**Configuration**	**Mean exec time (s)**	**Speedup**	**Efficiency**
500	30%	Serial	0.00821	1.00	1.00
		Parallel -n 2	0.009685	0.847703	0.423851
		Parallel -n 4	0.009905	0.828874	0.207219
		Parallel -n 8	0.011185	0.734019	0.091752
	60%	Serial	0.0169	1.00	1.00
		Parallel -n 2	0.017765	0.951309	0.475654
		Parallel -n 4	0.018085	0.934476	0.233619
		Parallel -n 8	0.020745	0.814554	0.101832
	90%	Serial	0.054555	1.00	1.00
		Parallel -n 2	0.0532	1.025470	0.512735
		Parallel -n 4	0.055625	0.980764	0.245191
		Parallel -n 8	0.089685	0.608296	0.076037
1000	30%	Serial	0.03505	1.00	1.00
		Parallel -n 2	0.032855	1.066809	0.533404
		Parallel -n 4	0.03502	1.000857	0.250214
		Parallel -n 8	0.039695	0.882983	0.110373
	60%	Serial	0.07776	1.00	1.00
		Parallel -n 2	0.073965	1.051308	0.525645
		Parallel -n 4	0.079455	0.978667	0.244667
		Parallel -n 8	0.084035	0.925329	0.115666
	90%	Serial	0.16982	1.00	1.00
		Parallel -n 2	0.16374	1.037132	0.518566
		Parallel -n 4	0.023361	0.726938	0.181735
		Parallel -n 8	0.34582	0.491065	0.061383

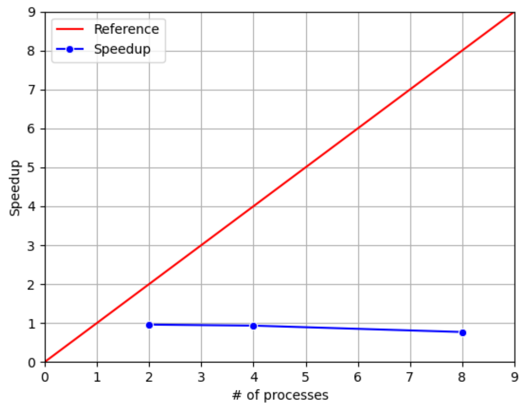
Table 2: Mean execution time, Speedup and Efficiency for OPT -O1

5.3 Optimization: -O2

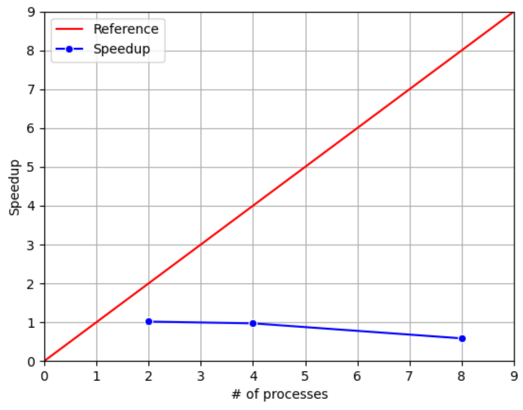
The -O2 level is the standard optimization level for production builds. It balances performance and reliability, executing a large number of optimization but not aggressive size-increasing or unsafe optimizations.



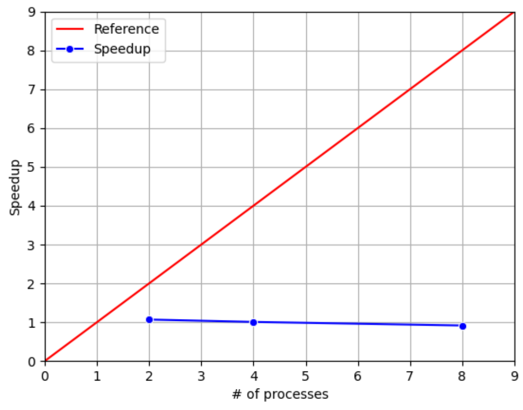
(a) 500 nodes and 30% coverage



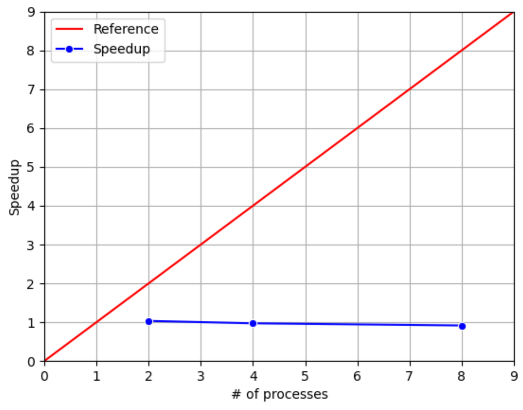
(b) 500 nodes and 60% coverage



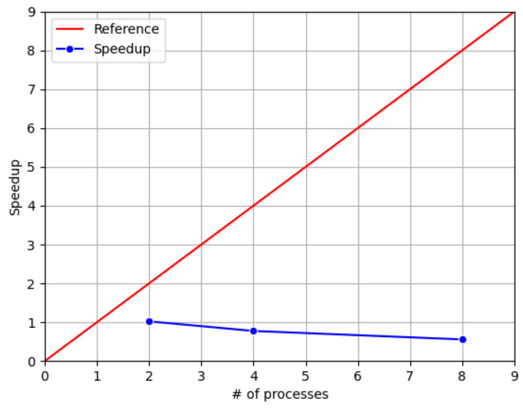
(c) 500 nodes and 90% coverage



(d) 1000 nodes and 30% coverage



(e) 1000 nodes and 60% coverage



(f) 1000 nodes and 90% coverage

Figure 7: Speedup for Optimization -O2

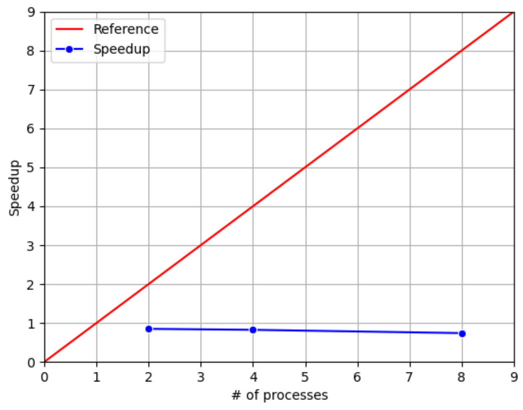
All the data acquired for this optimization are reported in the table below.

Nodes	**Coverage**	**Configuration**	**Mean exec time (s)**	**Speedup**	**Efficiency**
500	30%	Serial	0.008725	1.00	1.00
		Parallel -n 2	0.01018	0.85707	0.42854
		Parallel -n 4	0.010575	0.82506	0.20626
		Parallel -n 8	0.01128	0.77349	0.09669
	60%	Serial	0.0191	1.00	1.00
		Parallel -n 2	0.019925	0.95859	0.47930
		Parallel -n 4	0.020485	0.93239	0.23310
		Parallel -n 8	0.02483	0.76923	0.09615
	90%	Serial	0.06674	1.00	1.00
		Parallel -n 2	0.065645	1.01668	0.50843
		Parallel -n 4	0.06875	0.97076	0.24269
		Parallel -n 8	0.114165	0.58459	0.07307
1000	30%	Serial	0.039025	1.00	1.00
		Parallel -n 2	0.036515	1.06874	0.53437
		Parallel -n 4	0.03872	1.00788	0.25197
		Parallel -n 8	0.042625	0.91554	0.11444
	60%	Serial	0.095955	1.00	1.00
		Parallel -n 2	0.092785	1.03417	0.51708
		Parallel -n 4	0.098695	0.97224	0.24306
		Parallel -n 8	0.104705	0.91643	0.11455
	90%	Serial	0.2286	1.00	1.00
		Parallel -n 2	0.223685	1.02197	0.51099
		Parallel -n 4	0.295095	0.77467	0.19367
		Parallel -n 8	0.41	0.55756	0.06970

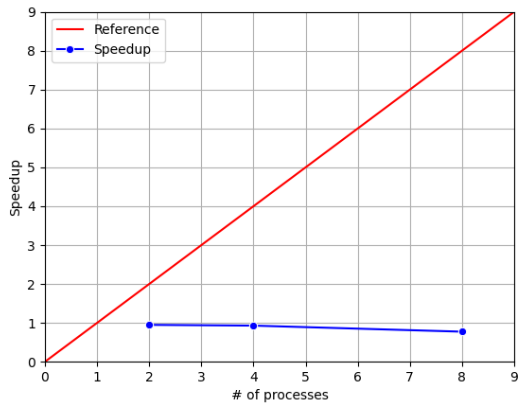
Table 3: Mean execution time, Speedup and Efficiency for OPT -O2

5.4 Optimization: -O3

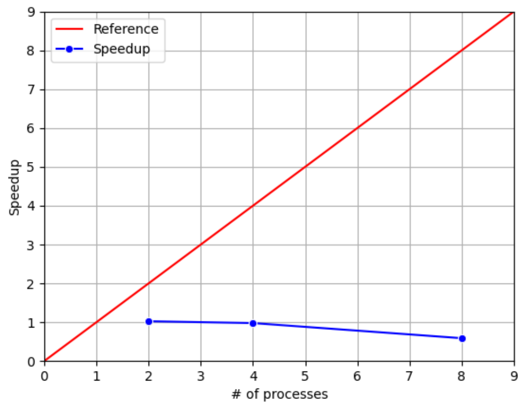
Lastly, -O3 is the most aggressive optimization level that enables aggressive optimizations focused on maximum execution speed, even if it increases code size and compilation time. This level attempts to make the compiled program run as fast as possible.



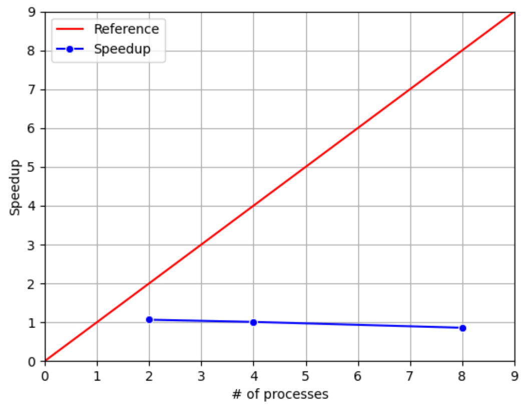
(a) 500 nodes and 30% coverage



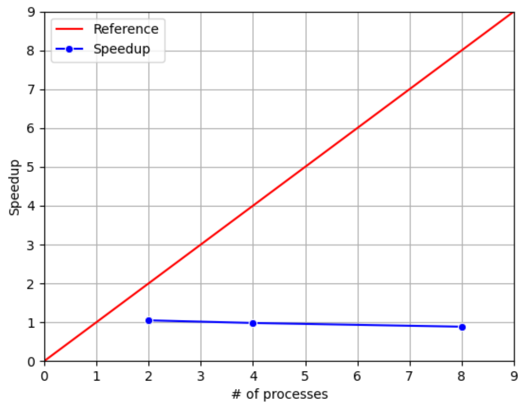
(b) 500 nodes and 60% coverage



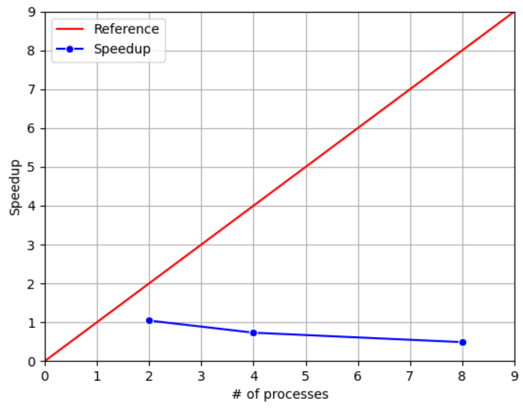
(c) 500 nodes and 90% coverage



(d) 1000 nodes and 30% coverage



(e) 1000 nodes and 60% coverage



(f) 1000 nodes and 90% coverage

Figure 8: Speedup for Optimization -O3

The data for this last optimization are collected in the table below.

Nodes	**Coverage**	**Configuration**	**Mean exec time(s)**	**Speedup**	**Efficiency**
500	30%	Serial	0.008075	1.00	1.00
		Parallel -n 2	0.009495	0.850448	0.425224
		Parallel -n 4	0.009785	0.825243	0.206311
		Parallel -n 8	0.010905	0.740486	0.092561
	60%	Serial	0.01662	1.00	1.00
		Parallel -n 2	0.01751	0.948886	0.474443
		Parallel -n 4	0.017875	0.929510	0.232378
		Parallel -n 8	0.02148	0.773510	0.096689
	90%	Serial	0.050945	1.00	1.00
		Parallel -n 2	0.04968	1.025463	0.512731
		Parallel -n 4	0.052105	0.977737	0.244434
		Parallel -n 8	0.086545	0.588653	0.073582
1000	30%	Serial	0.034485	1.00	1.00
		Parallel -n 2	0.03239	1.064680	0.532340
		Parallel -n 4	0.0342	1.008333	0.252083
		Parallel -n 8	0.0402	0.857836	0.107229
	60%	Serial	0.07641	1.00	1.00
		Parallel -n 2	0.07286	1.048724	0.524362
		Parallel -n 4	0.077925	0.989558	0.245140
		Parallel -n 8	0.086385	0.884529	0.110566
	90%	Serial	0.165265	1.00	1.00
		Parallel -n 2	0.158535	1.042451	0.521226
		Parallel -n 4	0.226255	0.730437	0.182609
		Parallel -n 8	0.338955	0.487572	0.060947

Table 4: Mean execution time, Speedup and Efficiency for OPT -O3

5.5 Considerations

Evaluating the plots and the data in the tables for speedup and efficiency, it may seem that the parallel version of the VF2++ algorithm performs worse than its serial counterpart. This is especially evident from the plots showing that, as the number of parallel processes increases, the overall performance often drops—sometimes significantly. The efficiency values in the tables confirm this: configurations with 8 parallel processes typically show the lowest efficiency, and in general, efficiency decreases with more processes.
However, these metrics do not account for a key factor: the number of graphs being analyzed simultaneously. If we assume that the execution time for each graph remains relatively constant and that graph comparisons are independent, we can estimate total serial execution time as:

$$TotalExecutionTime = SingleExecutionTime \times n. \tag{5.3}$$

For instance, analyzing seven graphs of 1000 nodes with 90% coverage and optimization -O3 takes 0.165 s per graph in the serial version. Thus:

$$TotalSerialTime = 0.165 \times 7 = 1.155s \tag{5.4}$$

The parallel version completes this task in just 0.339 s—significantly faster. This trend holds across various configurations. For smaller graphs (e.g., 500 nodes, 30% coverage, -O0 optimization), the serial version needs 0.98 s to analyze seven graphs, while the parallel version completes it in only 0.017 s.
The observed low efficiency is likely due to the Master process in the parallel implementation not participating in actual computation. Instead, it handles coordination and result aggregation. Additionally, inter-process communication introduces further overhead, especially as the number of processes grows.
In conclusion, although the efficiency and speedup metrics suggest suboptimal scaling, the parallel version still provides significant performance improvements for large-scale or batch analysis tasks.

Finally, all the data gathered, including all execution times for each run, are gathered in *.CSV* files in the "*Measures* folder inside each project. The mean time, speedup and efficiency values for all the configuration are available in the *.xlsx* file present in the root directory. Each directory containing code, including the two python script, is accompanied by a *REAMDE.txt* file for further explanation of the code and the parameters accepted by the command lines.

6. Possible future improvements: METIS and ParMETIS libraries

While developing the VF2++ C serial software, the hardest problem encountered was to decide how to divide the graph in n subgraph, smaller than the original one and possibly each with the same number of nodes. One partial solution to this problem was found in the library $METIS$[6] and it's MPI parallel version $ParMETIS$[7].

6.1 METIS

METIS is a C library provided by the George Karypis' research group at the University of Minnesota: it is a set of serial programs for partitioning graphs, partitioning finite element meshes, and producing fill reducing orderings for sparse matrices. The algorithms implemented are based on the multilevel recursive-bisection, multilevel k-way, and multi-constraint partitioning schemes developed in our lab.
However this solution uses a specific format for representing graphs, called "Compressed Sparse Row", or CSR. These representation is an efficient way to store sparse matrices, which are matrices with mostly zero elements, by only storing the non-zero values and their positions. It consists of three arrays:

- **value**: array used to store non-zero elements;
- **col_ind**: array used to store the column indexes of the non-zero elements;
- **row_ptr**: array used to store the starting index of each row in *values* and *col_ind*.

Of course this was a problem since the graph representation used in the software developed was completely different. A workaround was to implements several function to convert the graph used in the VF2++ algorithms in the CSR format, partitioning the newly crafted graph and converts each partition back into the original graph structure. After this process, the software could apply the VF2++ algorithm on subset of the original graphs and hopefully be faster in the execution (even with the addition overhead introduced by the conversion functions).
Unfortunately a new problem emerged after the partitioning process: partitioning a graph involves choosing if maintain the references to the nodes that are not in the subset or deleting this references. It's not a trivial decision since:

- keeping the references in the adjacency lists, although helps to maintain a structural integrity of the old, complete, graph, provide several out of bounds error, caused by the fact that not all the nodes in the adjacency lists were actually presents in the subgraph;
- deleting these references gave the software no method to keep track of which subgraph have already been explored, making impossible to account for eventual overlapping subgraphs.

As already said, the problem is not trivial and each solution seems to not bring to an optimal partition of the graphs, so the serial software kept its basic operation of comparing two graphs of equal size (equal number of nodes), not partitioned, while the MPI parallel version was instead focused on comparing several graphs, one for each process available.

6.2 ParMETIS

$ParMETIS$ it's the parallel METIS implementation, based on the MPI technology, developed by the same author of the METIS library. It's optimized for very large graphs, even shared on several machines, so it's perfect for HPC applications. Even though it is cited in the project, it's functionality has not been explored for the same problem described before.

List of Figures

List of Tables

Bibliography

[1] Wikipedia. URL: https://en.wikipedia.org/wiki/Graph_isomorphism.

[2] Alpár Jüttner and Péter Madarasi. “VF2++—An improved subgraph isomorphism algorithm”. In: *Discrete Applied Mathematics* 242 (Mar. 2018). DOI: 10.1016/j.dam.2018.02.018.

[3] techtarget. URL: https://www.techtarget.com/searchenterprisedesktop/definition/message-passing-interface-MPI#:~:text=The%20message%20passing%20interface%20(MPI)%20is%20a%20standardized%20means%20of,parallel%20program%20across%20distributed%20memory.

[4] Wikipedia. URL: https://en.wikipedia.org/wiki/OpenMP.

[5] MiviaLab - DIEM - UNISA. URL: https://github.com/MiviaLab/vf2lib.

[6] KarypisLab - UMN-CS&E - George Karypis’ research group. URL: https://github.com/KarypisLab/METIS.

[7] KarypisLab - UMN-CS&E - George Karypis’ research group. URL: https://github.com/KarypisLab/ParMETIS.