University of Basel

Optimizing Parallel Processes-to-Nodes Mapping in Contemporary High-Performance Interconnection Topologies

Master's Thesis

Faculty of Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
High Performance Computing
hpc.dmi.unibas.ch

Advisor: Prof. Dr. Florina M. Ciorba
Supervisor: Jonas H. Müller Korndörfer

Fatjon Lala
fatjon.lala@stud.unibas.ch
2018-063-412

17.12.2021

Abstract

The evolution of today's system of supercomputers has created two main issues: *the load imbalance* and the poor management of *data locality* of parallel applications. Therefore with the increase in the number of cores and the decrease of the memory size per core, performance is strongly correlated with load balancing and locality of data. In this thesis, we deal with the problem by considering the task-to-nodes mapping of parallel applications. Parallel application's performance is influenced by the mapping (known also as placement) of the processes onto the computing nodes, the frequency and volume of exchanges among the processing elements, the network capacity, and the routing protocol, among others. A poor mapping of application processes degrades performance and wastes computing resources and also increases energy consumption. Mapping processes in an application- and topology-aware manner is expected to minimize application performance degradation and optimize system resource usage. Both goals are critical for advancing scientific discovery and the efficient use of high-performance parallel and distributed computing systems. The mapping of an application is a well-known NP-complete problem, according to [19], therefore we will consider using heuristic strategies to achieve a sub-optimal solution. This master thesis focuses on using *LibTopoMap* [18][19], a generic mapping developed for arbitrary networks with the mapping library. A combination of this library with the algorithms from *MapLib* [31][27] guarantees a wide coverage of network topologies, mapping algorithms w.r.t the underlying architecture of the supercomputer. The extension of the libraries, therefore, grants the support for more mapping algorithms and more processor network topologies.

Acknowledgments

Throughout the writing of this dissertation, I have received a great deal of support and assistance. First and foremost, I am extremely grateful to Prof. Dr. Florina M. Ciorba for having me as a part of the High-Performance Computing (HPC) research team. She was always thriving to provide invaluable advice and continuous support during my thesis.
I would like to express my gratitude to my supervisor Jonas H. Müller Korndörfer for his constructive feedback, crucial insights, and patience during this work. His knowledge was a huge help to understand different scopes that triggered a out-of-box mentality on myself.
Furthermore, I would like to thank my father Fatos Lala, my mother Afërdita Lala, and my sister Fabjola Lala, for continuously encouraging and supporting me throughout my studies.
I am also grateful to my cousin Ervin Lala and his family (his wife Valbona, and his sons Enis and Eris), for supporting me during my initial and most difficult phase of the studies.
From the bottom of my heart, I thank everyone that made this work possible.
Finally, I would like to thank all my friends who were a great support throughout my studies and proofread this work.

Table of Contents

1 Introduction **1**

1.1 High-Performance Computing 1

1.2 Motivation 2

2 Background **3**

2.1 The mapping problem 3

2.2 Strategies for processes-to-nodes mapping 6

2.2.1 Greedy 6

2.2.2 Graph partitioning 6

2.2.3 Graph similarity 6

2.2.4 Graph isomorphism 6

2.3 Mapping enforcement techniques 6

2.3.1 Resource binding 7

2.3.2 Rank reordering 7

2.4 Interconnection network topologies 7

2.4.1 Direct topologies 7

2.4.2 Indirect topologies 8

2.5 Communication evaluation metrics 10

2.5.1 Physical communication metrics 10

2.5.2 Logical communication metrics 11

2.6 Tools 12

2.6.1 Graph partitioning software 12

2.6.2 Hardware information software 12

2.7 Programming frameworks 12

2.7.1 Message Passing Interface 12

2.7.2 Other programming paradigms 13

3 Related Work **14**

4 Methods **19**

5 Results & Discussion **22**

5.1 System 22

5.2 Design of experiments 22

5.3 Performance analysis 23

5.3.1 Experiments using two-level fat-tree network topology 23

5.3.2 Experiments using a file describing 3D torus network topology 28

5.3.3 Experiments using a file describing dragonfly network topology 29
5.3.4 Experiments using a file describing 3D mesh network topology 30

6 Conclusion & Future Work(s) 31
6.1 Limitations & challenges 31
6.2 Conclusion(s) 31
6.3 Future work(s) 32

Bibliography 33

Appendix A Appendix 36
A.1 Installing and understanding the libraries 36
A.1.1 LibTopoMap 36
A.1.1.1 Installation 36
A.1.1.2 Configuration files 37
A.1.1.3 LibTopoMap API 41
A.1.2 TopoMatch 43
A.1.2.1 Installation 43
A.1.2.2 TopoMatch API 44
A.1.3 MapLib 46

1. Introduction

In the first chapter of this work, we define the motivation, goals, and present a brief summary of this scientific work. We present some basic insights about high-performance computing (HPC) systems and their contribution to solving scientifically-hard computations, which simulate reality by using mathematical formulation.

1.1 High-Performance Computing

HPC is the ability of systems to perform complex calculations and process data very fast. These systems are made of supercomputers that execute tasks at high performance. Large-scale scientific applications such as weather prediction, computational chemistry, financial risk modelling, and machine/deep learning require a lot of computing power to be executed. Therefore the need for HPC systems is growing day by day. These systems have three main components:

1. Compute
2. Network
3. Storage

Compute nodes are connected in what is known as a cluster. These nodes are responsible for the execution of tasks. They communicate with each other using the network, which might be of different structures, and they use the storage to store the results coming from the execution of the application. As the number of nodes increases, also the complexity of the HPC systems increases. These nodes need to be configured to work as one entity during the computation.
The nodes, known also as hardware computing units (HCUs), execute the tasks in a distributed fashion and compute the results in parallel. To achieve the highest performance, the scientific applications are also designed and implemented with parallel architecture in mind. It is still ongoing work to achieve the best performance from the systems in proportion to the HCUs.
Today, different parallel programming paradigms exist. The most known of which is MPI [17] and OpenMP [10]. Message Passing Interface (MPI) is a communication protocol for programming in parallel computer systems. It creates the possibility for the parallel processes to use their memory and exchange messages to share the data. Open Multi-Processing (OpenMP) is a multi-threaded implementation technique that makes use of shared memory. It consists of threads, which use the same memory to access the same data. Hybrid implementation of MPI and OpenMP is possible, with MPI responsible for inter-node (node-to-node) and OpenMP for intra-node (in-node) parallelism.

1.2 Motivation

As we move towards the exascale era of supercomputers, we observe an increased number of HCUs and the number of cores per HCU. The increase of the computing elements, therefore, increases the complexity of the entire system. To handle the complexity problem of such systems we have to consider solving different problems, which among researchers are known as bottlenecks.

Once an application is designed to be performed in parallel different components should be taken into account. One of the most important ones is data decomposition. Data decomposition is the process to divide the application computation into smaller tasks that may be executed in parallel [16]. A programmer should take into consideration also the distribution of the input, output, intermediate data, the mapping of the task to software processing elements (SPEs), the synchronization of the processors that execute a parallel application, and so on.

The mechanism by which tasks are assigned to processes for execution is called *mapping* [16]. It is shown that the way how the tasks are mapped into the processes affects the overall performance of the system w.r.t an application. One of the key aspects that are shown to have a huge impact on the performance of the system is the communication between processes or also known as inter-node communication. When an application is decomposed, different parts of this application will be considered to be computed in different nodes. These parts of the application can have dependencies between each other or even in the best case, they have to come up with a mutual result. Sequential mapping is commonly used to map tasks to nodes for execution. This leads to a non-efficient usage of the components because the communication between SPEs is not considered while mapping. Therefore, parts of the application or widely known as tasks, that depend on each other are mapped into HCUs that might be located far from one another.

The optimization of the processes-to-nodes placement, by taking into consideration the processes communication, is the motivation for this work. The goal can be achieved by mapping tasks that are dependant on each other, to HCUs that are located closer to one another. In the following section, a more detailed description of the mapping problem is defined.

In this thesis, we enhance two libraries, to support more mapping algorithms and/or interconnection network topologies. A configurable point-to-point (P2P) communication application, named *COMAP*, is created. We make use of the strong sides of each library to achieve an optimized task-to-nodes mapping. We exploit the usage of LibTopoMap in the created application. MapLib is enhanced to support two indirect topologies: *dragonfly* and *two-level fat tree*.

The work is organized as follows. In Chapter 2, the background information about the problem is presented. The related work is carefully reviewed in Chapter 3. The methods used in this work are presented in Chapter 4. Results and discussion over them are shown in Chapter 5. Chapter 6 concludes the work and sets the path for future work(s) on this topic.

2. Background

This chapter provides fundamental information about the purpose of this work. It is compounded by different parts such as the mapping problem, evaluation metrics, strategies for process-to-nodes mapping, mapping enforcement techniques, interconnection network topologies, tools, and programming frameworks.

2.1 The mapping problem

The mapping problem can be considered as a graph embedding problem [6]. Moreover, the mapping problem can be seen as a minimization problem that tends to minimize different metrics [6]. Regarding this, process-to-nodes mapping is considered to be an important aspect of optimizing the efficient use of HPC systems. In the literature, two different types of mapping strategies have been mainly defined: 1)*topology-aware*, and 2)*topology-oblivious* mapping. Topology-aware process mapping consists of mapping the processes w.r.t the communication pattern (matrix) of an application, on the underlying hardware architecture. The topology-oblivious strategy takes into consideration neither the communication matrices nor the underlying hardware architecture [27]. The first approach is usually used to evaluate the mapping of a real-world scenario, where the underlying architecture has a lot of impact on the performance of the application. While the second approach is mainly used to evaluate the mapping strategies/algorithms while taking into consideration a broader number of interconnection network topologies. In both cases, an efficient mapping would result in preventing the communication between processes via slow network channels and would reduce the message's long transmission paths.
The communication of the processes in a parallel application can be represented as a graph, $G = \langle V, E \rangle$, where vertices V represent the set of processors while the edges E, which can be represented as $E \subseteq VxV$, denotes the communication pairs [6] [19] [5]. We use the logical communication representation to denote the volume of communication from process u to process v, and the physical communication which is known as the interconnection network topology [19]. The logical communication graph, is represented with $G = \langle V_G, w_G \rangle$, where V_G is a set of processes while w_G is the weight of the edge that connects two processes $u, v \in V_G$ [18]. On the other hand, the physical communication graph is denoted with $H = \langle V_H, C_H, c_H, R_H \rangle$, where V_H is a set of physical nodes (processing units or switches). If $u \in V_H$ then $C_H(u)$ is the number of processes that can be hosted at u (this represents multi-core processors); $C_H(u) = 0$ if u contains no processors (e.g., is a switch). $c_H(uv)$ is the capacity (bandwidth) of the link connecting u to v (zero if there is no such link). The function R_H represents the routing algorithm [19]. To understand better the mapping problem, let us assume that we have executed an application whose communication matrix, defined as a process graph, is shown in Figure 2.1.

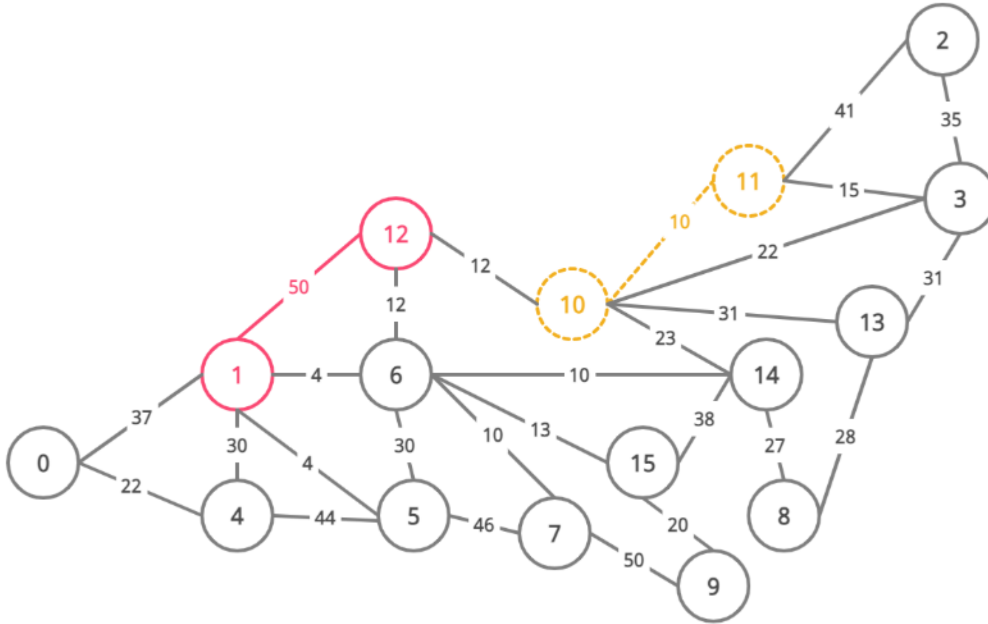

Figure 2.1: Process graph of an application.

The process graph (Figure 2.1) presents the SPEs that communicate with each other (as vertices connected by an edge) and the volume of their communication (as edge weights). As we can see, process 1 communicates with process 12, with a communication volume equal to 50. In this example, this is considered intensive communication. On the other side, process 10 communicates with process 11 with a communication volume equal to 10, which depicts a lower communication intensity. The process graph is overlaid onto what is known as a systems' graph. It is a representation of the interconnection network topology of a system. Figure 2.2 shows the system graph of miniHPC [3] that is the system used for the experiments in this work.

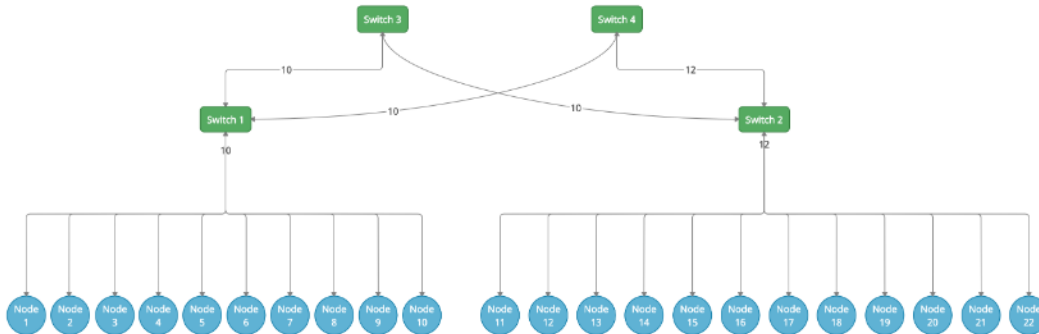

Figure 2.2: System graph of an interconnection network topology (miniHPC [3]).

The system is compounded from 22 computing elements (nodes), and 4 routing elements (switches). The number of links, between the switches and nodes, are shown in the bi-directional links.

So far, we presented the process graph of an application and the graph of a system. The default (initial) mapping of processes to nodes is achieved from the scheduler of the operating system. It consists of one-to-one mapping of processes to nodes, in a sequential increasing order of ids, which is presented in Figure 2.3. We use the term **rank** to refer to a process.

Figure 2.3: Default mapping from the scheduler of the operating system.

In Figure 2.3 we can see that each process (MPI rank) is assigned to a node. Since we had 16 processes in the process graph, 16 nodes are occupied and 6 nodes are still free on the system (grey-coloured). Rank 1 and 12, according to the process graph in Figure 2.1, communicate intensively with each other but still, they are mapped on nodes physically far from one another. On the other side, rank 10 and 11, representing a low-intensity communication (Figure 2.1), are mapped on nodes closed to each other. Logically, the time needed for the communication in the application would be high as well as the execution time. The default mapping can be optimized by using different mapping strategies (more details in Section 2.2). In Figure 2.4, the optimized mapping derived from the greedy mapping algorithm is presented.

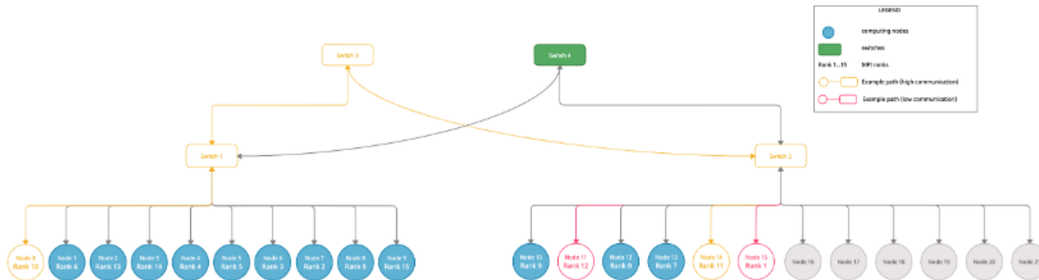

Figure 2.4: Optimized mapping with greedy mapping strategy.

As we can see from Figure 2.4, rank 1 and 12 are mapped to nodes that are close to each other. While on the other hand, rank 10 and 11, are mapped to nodes that are far from each other.
With this example, we wanted to visually present the mapping problem and how it is treated in this work. In the next section, we provide some more information about the mapping strategies that are used during the experimentation phase of this thesis.

2.2 Strategies for processes-to-nodes mapping

Hoefler et al., in [19], show that the mapping problem is NP-complete. This means that the problem cannot be approximated well [20]. Therefore to deal with the problem and to be able to come up with a sub-optimal solution different strategies have been followed. In this thesis, we distinguish different strategy categories such as 1)*greedy*, 2)*graph partitioning*, 3)*graph similarity*, and 4)*graph isomorphism*.

2.2.1 Greedy

It is one of the most common heuristics used to solve graph problems. The algorithm starts with a vertex in H (known as the physical communication or systems' graph) and maps it greedily starting with the heaviest vertex in G (known as the logical communication or process graph). It continues to map the heaviest vertices in H and G together until it reaches the end of the execution. It gives priority to the vertices that communicate more with each other and it does this recursively by using a greedy approach. Some of the algorithms that fall into this category are Peano, Hilbert, Gray, sweep and scanSFC, greedy, FHGreedy, greedyALLC, and topo-aware which are supported from MapLib [27].

2.2.2 Graph partitioning

Another strategy that we use in this work is graph partitioning. It basically, cuts the physical (H) and logical (G) communication graph into smaller parts. In this work, we consider bi-partitioning, which means that we divide the graph into two equal halves recursively by determining the minimum over edge weights. This method is also known as bisection mapping. Another algorithm that falls into this category is PacMap[33].

2.2.3 Graph similarity

Graph similarity approach, used to map processes to nodes, is a well-known technique. The basic idea is to bring adjacency matrices between the system and process graph into a similar shape. Reverse Cuthill McKee (RCM) [9] algorithm is used to solve the reduction of bandwidth problem in a heuristic way.

2.2.4 Graph isomorphism

Graph isomorphism, in terms of the mapping problem, is just the one-to-one correspondence between the edges and the vertices of the physical (H) and logical (G) communication graphs. The algorithm that falls into this category is known as Bokhari algorithm [6], which corresponds to the first research study on this topic.

2.3 Mapping enforcement techniques

During the mapping process, the processes are assigned to computing units. Based on the mapping algorithm and application we apply a certain placement policy. There are different techniques to enforce such policy, but we present two of them: 1)*resource binding*, and

2)*rank reordering*.

2.3.1 Resource binding

Resource binding is a technique to bind SPEs to HCUs [20]. In different systems, different commands are used to do this work. The only issue is portability across different systems and architecture. To solve this issue we use HwLoc [7], a tool used to gather the hardware information which is not dependant on the system that is being used. By doing so, we actually tell the operating system which processing element would belong to which hardware unit. If we leave the operating system to decide, then we might have cache misses and therefore we cannot get the most out of the system. Even though binding is a process that is not correlated with the programming framework, so it is outside of the implementation, it can affect the performance of the systems for the given applications.

2.3.2 Rank reordering

The second technique which is commonly used among researchers is *rank reordering*. Indifference to resource binding, it does not tell the operating system what to do but it just changes the ids of the SPEs. Therefore, rank reordering is not related to any tool because it can be defined within the programming framework and we can see its common usage in MPI. That is the main reason why we consider rank reordering because it gives us the flexibility to work on different systems without having to worry about the operating system scheduler or the underlying architecture. Even though, according to [20], a poor scheduling decision effect can apply also for the application using only rank reordering. We assume that the interference of the operating system's scheduler will be minimal but it might be a very useful study case to be considered as future work.

2.4 Interconnection network topologies

In this section, we give some information about the different network topologies that we use in this work. We categorize these topologies, based on their communication style, into two main groups, *direct* and *indirect* topologies. In direct network topologies, nodes are connected directly with their neighbouring nodes network interface card (NIC). While in the indirect network topologies, nodes are not connected directly to each other but are connected with switches, which are responsible for the information routing through the network from one node to another.

2.4.1 Direct topologies

In this work we consider two direct network topologies to run our experiments. The first topology is 3-dimensional (3D) torus, where one node is connected directly with its 6 neighbours. A graphical illustration of such network topology is used shown in Figure 2.5

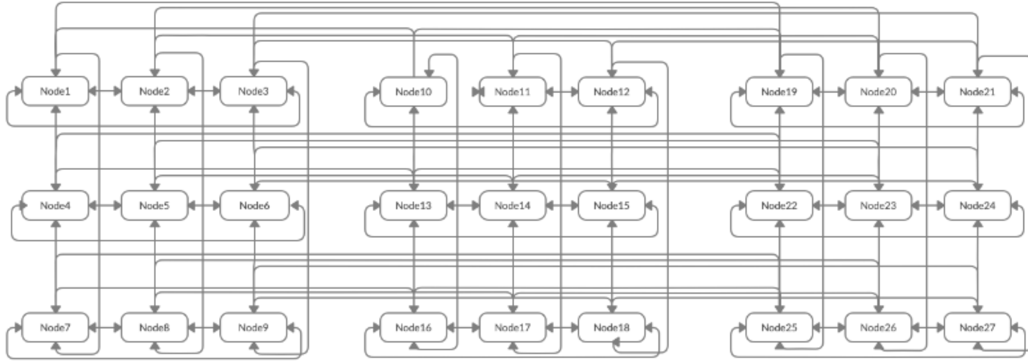

Figure 2.5: 3D torus network topology (2D representation)

The second direct topology that we consider in this work is a 3-dimensional (3D) mesh. This topology is similar to torus but it differs from the number of connections that each node has. Most of the nodes, or better saying 2/3 of them, connect directly with 4 other nodes. In contrast, 1/3 of the nodes or the nodes located in the middle, connect with 5 other nodes. In Figure 2.6, we can see a visualization of the topology.

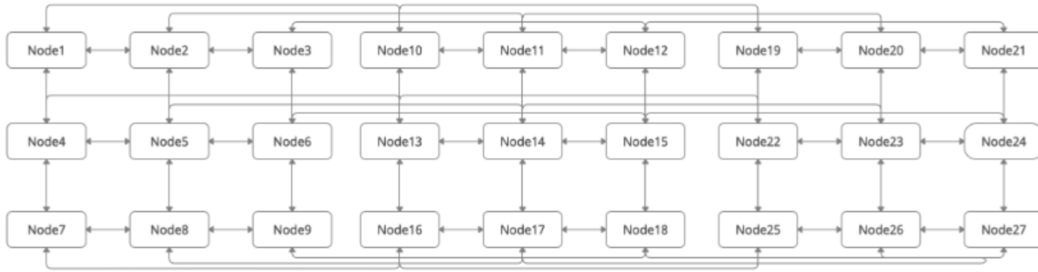

Figure 2.6: 3D mesh network topology (2D representation)

2.4.2 Indirect topologies

Regarding the indirect topologies, in this work, we consider *two-level fat tree* and *dragonfly* topology. Two-level fat-tree network topology is non-blocking (due to the equality of input and output channels for each switch) and is the same used in miniHPC [3], which is the cluster of the HPC research team at the University of Basel. The *storage*, *login* and *knights landing (KNL)* nodes are not considered in the experimental setup. More detailed information about the experimental setup and systems will be described later in this work. In Figure 2.7, it is shown the two-level fat-tree network topology that we have used for our work.

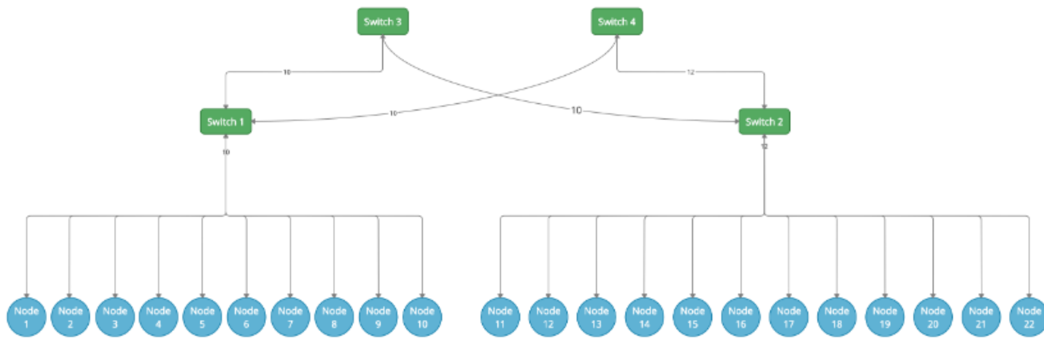

Figure 2.7: Two-level fat-tree network topology

As we can see in Figure 2.7, this topology is compounded from 26 elements. These elements are divided into 22 computing nodes (HCUs) and 4 routing elements (switches). In the representation, each node is connected directly with the switches (either switch 1 or 2), but for visibility purposes the connections are grouped. The other indirect network topology that we consider in this work is *dragonfly* [26], presented in Figure 2.8.

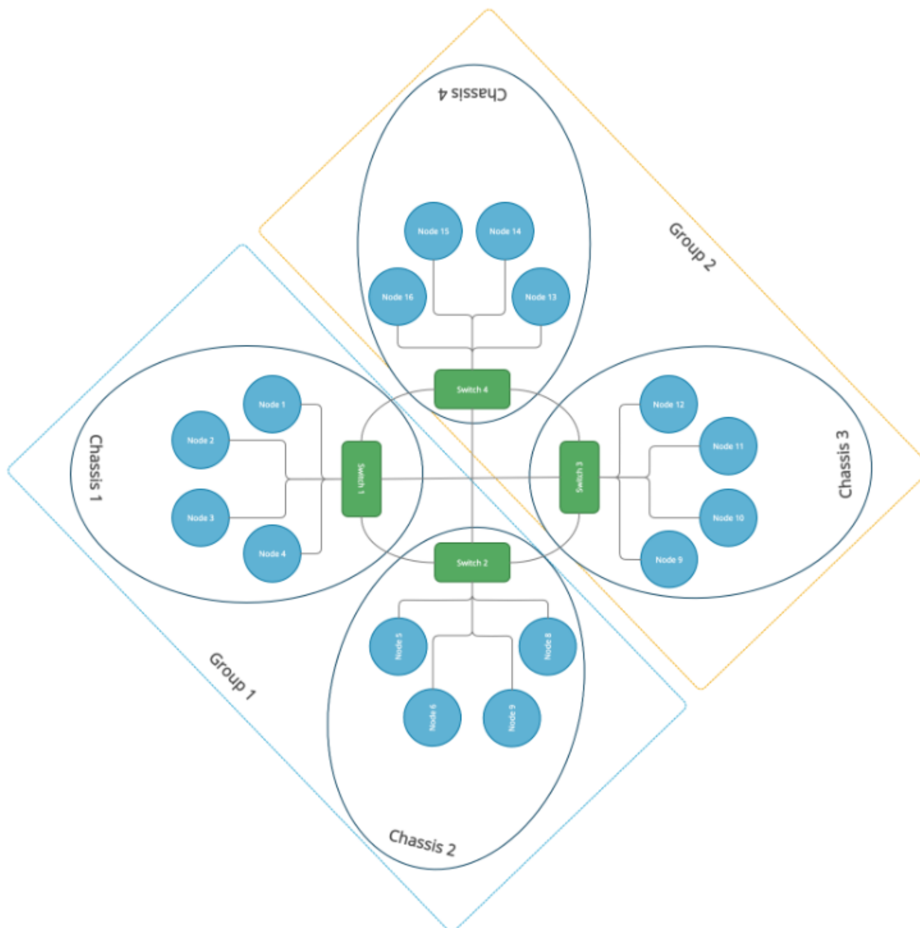

Figure 2.8: Dragonfly network topology.

This network topology uses a group of high-radix routers as virtual routers to increase the radix effect on a network [26]. It is a hierarchical network with three levels: *slot*, *chassis* and *group*. It consists of 2 groups, 2 chassis, and 5 slots per chassis. The composition of the slots in each chassis is achieved from 4 computing nodes and 1 switch. The connection in the topology is switch-based, where all the switches are connected to each other, via inter-chassis and inter-group connection links.

2.5 Communication evaluation metrics

This section is dedicated to the metrics used to evaluate the performance of the processes-to-nodes mapping achieved by different heuristic strategies. We consider two categories of these metrics:

1. Physical communication metrics.
2. Logical communication metrics.

2.5.1 Physical communication metrics

The most used communication evaluation metrics in the literature are *congestion* and *dilation* [19][21][27]. Dilation is mainly defined as the maximum or the sum of the pairwise distance of neighbours from H to G [27][20]. While congestion represents the communication pairs that use a certain link in the interconnection network. The main goal is to reduce the maximum congestion or better known as worst-case congestion. Also, a lower dilation value leads to improved performance and implicitly to a communication energy consumption reduction [27]. As we could show in the previous sections, processes-to-nodes mapping takes into consideration two types of communication patterns. The logical communication pattern is a weighted, directed graph that defines the volume of communication between processes (by the weight of the edge when the communication occurs) or 0 when the communication does not occur [18]. The physical communication pattern represents the network interconnection of physical nodes (processors and switches) [18]. Based on these communications, the calculations for the two main metrics, *dilation* and *congestion*, are computed.

- **Dilation(uv)**: the average length of the path taken by a message sent from u to v [18].

$$Dilation(uv) = \sum_{u,v \in V_G} R_H(\Gamma_{(u)}\Gamma_{(v)})(p) \cdot \mid p \mid \tag{2.1}$$

  where:

  $R_H(\Gamma_{(u)}\Gamma_{(v)})(p)$ fraction of the traffic from u to v that is routed through p when a mapping exists
  $\mid$ p $\mid$ length of the path p
  Γ function that maps $V_G \rightarrow V_H$

There are different calculations for *dilation* while in this work we use those defined in Equation 2.1 which is the same used in LibTopoMap. This metric can be also known

as the number of hops. This is an important aspect to be pointed out because it might lead to confusion among researchers. To solve this confusion, we use the term *dilation* for both notations, as suggested in LibTopoMap.
Congestion of a link uv of the interconnection network is the ratio between the amount of traffic on that link and the capacity of the link.[18]

- **Congestion(uv)** is computed by the traffic on the link and its capacity as follows:
  - **Traffic(uv)**:

$$Traffic(uv) = \sum_{u,v \in V_G} w_G(uv) \cdot \left(\sum_{p \in P(\Gamma(u)\Gamma(v)); e \in p} R_H(\Gamma(u)\Gamma(v))(p) \right) \tag{2.2}$$

where:

$w_G(uv)$ the weight of the edge connecting u to v. Represents the volume of communication from u to v (0 if there is no connection)
$R_H(\Gamma_{(u)}\Gamma_{(v)})(p)$ fraction of the traffic from u to v that is routed through p, when a mapping exists
Γ function that maps $V_G \rightarrow V_H$

  - **Congestion(uv)**:

$$Congestion(uv) = \frac{Traffic(uv)}{c_H(uv)}$$

where:

$c_H(uv)$ capacity of the link connecting edge *(*uv)

In this thesis, we will consider the worst-case congestion. The maximum overall congestions for each vertex of the graph is considered as congestion results. The same measure is also done when reordering the communicator (if the worst congestion of the initial mapping is higher than the worst congestion of the reordered communicator).

$$Congestion(\Gamma) = max_{uv} Congestion(uv) \tag{2.3}$$

2.5.2 Logical communication metrics

In this subsection, we present the measurements that can be done to the communication that happens between SPEs used during the execution of the application. Some of the metrics used in this work, which are also used in MapLib [31][27], are:

- Inter-process logical communication (IePLC), which defines the number of exchanged messages between any two SPEs. This metric is independent of the mapping strategy and is a property of the process graph. In LibTopoMap, it is considered to be the weight of the edges in the process graph.

- Intra-node logical communication (IaNLC) shows the total number of messages exchanged between processes that are mapped onto the same physical node. We calculate this value, using LibTopoMap, as the sum of exchanged messages between processes that are mapped onto the same physical node.

- Inter-node logical communication (IeNLC) is the sum of all exchanged messages between nodes resulting from the mapping of the communication pairs but neglecting the network topology. In LibTopoMap, this metric corresponds to P2P exchanged messages divided by the number of HCUs.

2.6 Tools

In this section, we give some information about the tools that are used in this thesis. We divide these tools/software into two main categories: *graph partitioning* and *hardware information software*.

2.6.1 Graph partitioning software

The first type of software considered in this thesis are *graph partitioning software*. They are mainly used to partition graphs which would make it easier to solve scientific computing problems in parallel. After the partitioning of the scientific computing problems as graphs then we can apply different heuristics to solve the problem sub-optimally. METIS [24] and ParMETIS [30] (its parallel version) are considered in this thesis. In difference to METIS, ParMETIS uses parallel multilevel k-way graph partitioning technique based on a MPI implementation. Another graph partitioning software that we use is SCOTCH [29]. This software deals with tree-structured data to perform the mapping [20]. It is also based on dual recursive bi-partitioning.

2.6.2 Hardware information software

As briefly mentioned before, in this category we make use of the Portable Hardware Locality (HwLoc) software [7] to gather the hardware information about the underlying architecture. Therefore, we guarantee the portability of our solution, since HwLoc is not architecture- or system-bounded. It is a software that solves the issue of scheduling policy of the operating system. It provides a portable abstraction (across operating systems, versions, architectures, etc.) of the hierarchical topology of modern architectures, including NUMA memory nodes, sockets, shared caches, cores, and simultaneous multi-threading [1].

2.7 Programming frameworks

In this section, some information about the most common programming frameworks used to solve the mapping problem is presented. We use the MPI framework for this thesis, but some basic information about the other existing paradigms is presented in the following subsections.

2.7.1 Message Passing Interface

The Message Passing Interface (MPI) is a communication protocol that relies on distributed memory model connected with a flat network [14]. It is designed for parallel computing sys-

tems which address the problems of scalability, high performance and portability. It is used widely on certain classes of parallel machines, especially those with distributed memory [14]. In this work, we work with MPI 2.2, which supports the virtual topology building and management. It provides two variants of distributed graph constructor interface: *MPI_Dist_graph_create* and *MPI_Dist_graph_create_adjacent* function. The first function is general while the second one is an adjacent specification. Both functions are collective, meaning that all processes in the old communicator must perform the call. We use the first variant because we are focused on a generic approach to deal with virtual topologies.

2.7.2 Other programming paradigms

Another programming paradigm that deals with the issue of virtual topologies is CHARM++ [23]. It enables users to easily expose and express much of the parallelism in their algorithms while automating many of the requirements for high performance and scalability. It permits writing parallel programs in units that are natural to the domain, without having to deal with processors and threads. CHARM++ rather uses finer-grain objects and dispatches the computation onto small migratable tasks called chares [22]. These chares, in addition to their assigned data and their ability to exchange messages asynchronously, are also characterized by their CPU load, their I/O communication volume and some other useful parameters [22]. In our implementation, we do not consider using CHARM++ but it might be an interesting paradigm to be considered as future work.

3. Related Work

Topology mapping problem is an interesting topic among researchers. Different researches go beyond on-node mapping and consider also the in-node mapping as a second scale for performance gains. Therefore, the complexity of parallel application can be efficiently exploited with the right task mapping algorithm.
The initial study for the problem was made from Bokhari et al.,[6], where the task to nodes mapping was considered as a graph embedding problem. By using this approach, a mapping algorithm called *MAPPER*, a pairwise interchange algorithm, was introduced.
Communication- and topology-aware mapping.
MAPPER [6], is an algorithm that tries to solve the mapping problem while considering the topology of the system. It consists of calculation done into two main steps. In the first step, it considers all possible pairs of nodes that communicate with each other in the system. The nodes are considered pairs if their communication gives the highest cardinality of the mapping graph. In the second step, some random (probabilistic) jumps are made to solve the issue of the dead-ends. If a better mapping is found during the random jumps from the algorithms, then the first step is being called, otherwise the algorithm terminates. The algorithm employs *rank reordering* technique to do the mapping and tries to permute the adjacency matrix of the graph problem that matches more closely the adjacency matrix of a Finite Element Machine (FEM). This machine is an array of microcomputers interconnected in an *eight-nearest neighbor* interconnection pattern [6]. Bokhari et al., consider the direct interconnection network topologies with different sizes for their experiments.

Hoefler et al., [18] present LibTopoMap, a generic mapping library. Their approach consists of three phases: 1)*gathering the applications communication information among processes*, 2)*defining the architecture of underlying hardware*, 3)*computing and enforcing the process placement*. To create the graphs from the application communication pattern and the underlying hardware architecture different partitioning libraries are used. LibTopoMap uses METIS [30] and/or SCOTCH [29] to achieve an initial optimized partitioning and ParMETIS [30] (parallel version of METIS) to balance the partitions if necessary. It is worth mentioning that the gathering of the applications communication pattern and the construction of the architecture of the underlying hardware needs to be done by the programmer explicitly.
Communication- and topology-aware mapping.
Hoefler et al., make use of different algorithms such as: *greedy*, *recursive bisection*, *Reverse Cuthill-McKee (RCM)*[9] and *Threshold Accepting (TA)*. *Greedy* starts the mapping of a random process to a random node and it maps recursively the heaviest communicating processes to nodes that are physically close to each other. The other algorithm, *recursive bisection mapping*, is based on splitting the minimum weighted edge-cut of the logical and physical communication graph. The recursive bisection mapping is considered as graph partitioning strategy, shown in Table 3.1. The *Reverse Cuthill-McKee (RCM)* [9], is a heuristic

used to solve the bandwidth reduction problem, which is a problem equivalent to the mapping problem according to [6]. RCM turns, the adjacency matrices between the physical and logical communication graphs, into similar shape, to achieve the mapping. The last algorithm consider in LibTopoMap is known as Threshold Accepting(TA) [12]. It is used to optimize the initial solutions achieved by the mapping algorithms and to solve the problem of local minimas. They also show that the benefit of topology mapping grows with the network size. Their mapping strategies have shown to reduce network congestion up to 80%, reduce average dilation up to 50%, and improve benchmark communication performance by 18%. LibTopoMap has place for improvement regarding the mapping algorithms, which is considered in this thesis. On the other hand also the network topologies are programmer-dependant, which means that the programmer is responsible for gathering the underlying architecture information, and we would improve it by using HwLoc [7] (which is a software that can retrieve the underlying architecture for us). To support LibTopoMap with algorithms, we use MapLib [31][27], which is a library that provides different mapping algorithms that would enhance the mapping abilities of the LibTopoMap library.

In [21], Jeannot et al., propose a distributed library called TopoMatch, which is an extended version of TreeMatch.

Communication- and topology-aware

Its main mapping algorithm is TreeMatch, which uses a tree for modelling the hardware. There are two main steps to generate a mapping, which consist of: 1)*generating all possible combinations for a given set of processes (tasks)* and 2) *selecting the best independent combination*. The first step, creates pairs of processes that might have dependencies among each other. In the second step, the pair processes that have the lowest amount of communication reduced, are combined together, greedily, as the best independent combination. In TreeMatch, the topology tree is processed upward and processes are recursively grouped according to the arity of the next considered level. To avoid the combinatiorial explosion of the algorithm runtime, two mechanisms are introduced, namely: *arity division of the tree* and *speeding up the group building* [21]. Arity division of the tree is used to decompose a level of the tree into several levels, which then consists on a speed improvement of group building procedure and search simplification due to the fact of have large number of groups. Another algorithm considered in TopoMatch is *greedy k-partitioning*, which is an extended greedy version of the k-way partitioning algorithm used in METIS [24]. It is shown that with different mapping techniques, 25% gain in communication time is being achieved. TopoMatch uses a three step approach for achieving the remapping, which are the same steps as the ones presented for LibTopoMap with small differences. HwLoc is used to gather the hardware information and TreeMatch algorithm is used to compute the reordering. This library is distributed which means that each node reorders only its local MPI processes.

In [27], Korndörfer et al., present a library of mapping algorithms, namely *MapLib*. This library consists of mapping algorithms from the literature, and generates mapping for 3-D topologies such as: *mesh*, *torus*, and *HAEC Box* [13].

Communication- and topology-oblivious mapping.

Five algorithms considering the space filling curves (SFCs), are used in MapLib. This algorithms are: *Paeno*, *Hilbert*, *Gray*, *sweep* and *scan*. These algorithms map multi-dimensional spaces onto one-dimensional spaces [28][27]. Sweep is used a default reference mapping to evaluate the quality and performance improvement achieved by all other mappings in MapLib [27]. In Table 3.1, you can see that all these algorithms (SFCs) are greedy algorithms based on the categorization of strategies for processes-to-nodes mapping defined in Section 2.

Communication- and topology-aware mapping.

MapLib makes use also of algorithms that take into consideration the interconnection network topology. The first one is *topo-aware* [4], which groups the intensive-communicating processes together in one task. As a second step, the algorithm maps these tasks to nodes on the underlying hardware w.r.t their communication intensity. The tasks that communicate more together are placed onto processors that are physically closer. Greedy algorithms are also considered in MapLib, specifically: *greedy* [18] (also considered in LibTopoMap), *FHgreedy* [11], and *greedyALLC* [15]. *Greedy* starts the mapping of a random process to a random node and it maps recursively the heaviest communicating processes to nodes that are physically close to each other. *Fhgreedy* start also by selecting a random process and mapping it to a node but it takes into consideration the amount of communication that processes have with each other. If two processes communicate often with each other then they are mapped in neighbouring nodes. While *greedyALLC* initially makes pairs of intensive-communication processes and uses these pairs to map them to nodes. Another type of mapping considered in MapLib is *bipartition*. It is a recursive solution to improve the node mappings on torus and mesh topologies [34]. It makes use of the directive of *k-way partitioning* (introduced in [25], to divide the application communication graph and the topology into two parts (bipartition = two-partitions). The last algorithm supported in MapLib is, Partitioning and Center Mapping (*PaCMap*) [33]. It is a graph-based algorithm that simultaneously carries out job allocation and task mapping to reduce communication overhead [33]. Firstly, it partitions the communication graph into k process groups where the size of the group is selected by considering the nodes of the cluster. METIS [24] *k-way partitioning* algorithm is used to achieve the partitioning while also in PaCMap processes that highly communicate with each other are mapped to the same process group. In the next step, PaCMap selects a center process group from the already partitioned graph and maps it to the selected center node in the cluster [33]. The algorithm finishes when all the tasks are mapped based on the interconnection network topology.

Somroo et al. [32], present BindMe, a thread binding library, which tries to determine the best mapping policy for the tasks of an application to the nodes of the architecture. It is a similar work to ours but it just considers the tree-like network topologies, while we will consider any network topology with any underlying architecture. It consists of getting the communication matrix of the application and applying different mapping algorithms to improve the communication time between cores.

Communication- and topology-aware mapping. They use different algorithms to achieve the mapping. The first one is *TreeMatch* [21] algorithm that is used in TopoMatch.

EagerMap [8] is another greedy algorithm used to map processes-to-nodes in BindMe library. It creates pairs of processes based on the intensity of the communication that the processes have with each other. In the next step, a tree topology is used to determine the group of processes in each level. This procedure is also applied to generate a mapping for a given interconnection network topology in EagerMap. *ChoiceMap* uses the application communication matrix and the interconnection network topology of the underlying architecture to determine the mapping which treats every process equally. Every process has its own choice list which is based on the communication intensity. The processes are paired by considering their nearest and mutual prioritized choices.

Characteristics \ Library name		LibTopoMap	TopoMatch	MapLib	BindMe
Strategies for process-to-nodes mapping	Greedy algorithm(s)	Greedy heuristics	Greedy k-partitioning Exhaustive search	Peano, Hilbert, Gray, sweep, scanSFCs, greedy, FGgreedy, greedyALLC, topo-aware	TreeMatch EagerMap
	Graph partitioning	Recursive bisection	Greedy k-partitioning	Bipartition PacMap	—
	Graph similarity	Reverse Cuthill-McKee (RCM)	—	—	—
	Graph isomorphism	—	—	Bokhari [1]	ChoiceMap
Interconnection network topologies	Direct topologies	Arbitrary	Arbitrary	3-D mesh 3-D torus 3-D HAEC Box	Arbitrary
	Indirect topologies	Arbitrary	Arbitrary	—	Arbitrary
Metric(s)		Congestion Dilation	Sum of all communications Maximum of all communications Dilation[2]	Dilation	Communication reduction value[3] Execution time
Tools/Software	Graph partitioning	METIS ParMETIS SCOTCH	TreeMatch SCOTCH	—	—
	Hardware information	—	HwLoc	—	Numalize HwLoc
Mapping enforcement techniques	Resource binding	no	yes	no	yes
	Rank reordering	yes	yes	yes	yes

Table 3.1: Characteristics of existing libraries.

Table 3.1 contains information about the characteristics of the existing libraries. It includes the information about the different types of strategies used for process-to-nodes mapping, interconnecting network topologies, metric(s), tools or software and mapping enforcement techniques as defined in the background section.

This work consists of making use of two different libraries (LibTopoMap and MapLib) and their characteristics, to come up with an enhanced version of these libraries. Therefore, this work makes the following contributions:

- Provides a configurable P2P communication application, named *COMAP*.

[1] *Bokhari* is the same algorithm named *MAPPER* in [6].
[2] Dilation is measured as hop-Byte, which is the product of the size of all the communications times their respective number of hops once mapped.
[3] Communication reduction value is used as a locality measure. It represents the amount of communication that the thread pair has to perform with all the other threads.

- Use LibTopoMap, as a generic approach, to map COMAP.
- Enhance LibTopoMap with different network topologies.
- Evaluate the performance measures of COMAP on LibTopoMap.
- Enhance MapLib to support more network topologies.

4. Methods

In this work we use LibTopoMap, as a generic mapping library, to map processes to nodes. LibTopoMap provides a package to be downloaded which includes the mapper and an example application (*reader.cc*). In Appendix A.1.1.1 we show the steps on how LibTopoMap can be installed. The example application (*reader.cc*) considers two parameters as input. These two parameters consist of a matrix file (in compressed sparse row (CSR) matrix, taken from SuiteSparse [2] matrix collection) and a file describing the interconnection network topology (as shown in Figure A.1, A.2, A.3, A.4). Initially, the matrix file is read by *reader.cc*. It decomposes the data to be processed from each SPE based on indexes. ParMETIS [25] is used to partition the data loaded on each SPE. With this information, a distributed graph communicator is created which is then used by LibTopoMap, alongside the topology file, to map the processes to nodes with different mapping strategies. In case a better (lower maximum congestion) mapping of SPEs to HCU is found, LibTopoMap returns a new identifier for each SPE (MPI rank).

During the initial phase of this study, we observed that LibTopoMap maps the processes of the distributed graph communicator into the nodes defined in the network file. The communication of the processes is necessary to create a distributed graph communicator. It includes the information about the neighbour, the number of neighbours, and the volume of communication for each process. For this reason, we created COMAP, a configurable P2P application. COMAP takes into consideration different parameters that configure different aspects of the communication. It considers *blocking* or *non-blocking* communication method. Furthermore, one can define the *number of exchanges* and/or *number of messages*. Lastly, it supports two types of communication patterns, *nearest neighbour (NN)* and *odd-even (OE)*.

COMAP accepts the parameters as integers (eg., 1, 2, 3, ...). The first parameter is the method of communication, 1 for blocking and 2 for non-blocking communication. The second parameter defines the number of exchanges which can be a value from 1 to the number of desired exchanges. The third parameter defines the number of messages. It can be a number from 1 to the number of desired messages to be exchanged. The last parameter defines the type of communication which is 1 for NN or 2 for OE. It is important to know that to be able to use the application with LibTopoMap, we need to add another parameter. This parameter is the file that defines the network topology of the system. In Figure A.1, an example of the two-level fat-tree topology file is shown. An example of the execution command of COMAP is shown below.

```
./COMAP physicalTopology.map 1 100 100 1
```

Sometimes, it is not possible to run the mapper on the target system. Therefore, LibTopoMap supports a simulation mode, where the user specifies a mapping of processes to nodes in a

so-called "fake file". This means that the naming of the nodes is not checked from the system but from another file. Some examples of such "fake files", known also as node naming file, can be found in Figure A.2 (a), A.3 (a), and A.4 (a). An example of the execution command of COMAP, while using the simulation mode, is shown below.

```
./COMAP physicalTopology.map 1 100 100 1 physicalTopology.fake
```

In Figure 4.1, we present the execution steps we followed during this work.

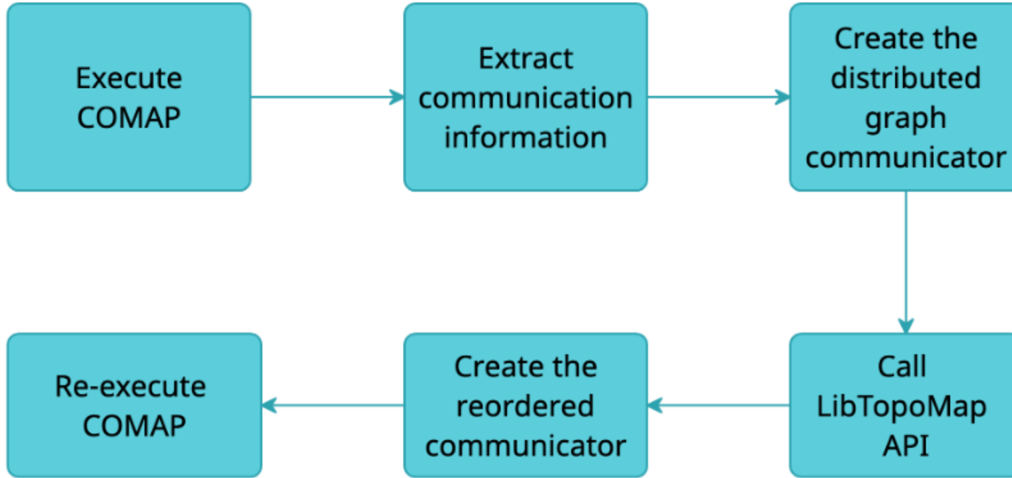

Figure 4.1: Execution steps

During the execution of the application, we extract the communication information. This information consists of the processes neighbours, the number of neighbours, and the volume of the communication for every process. At the end of the communication phase of the application, we create a distributed graph communicator with *MPIX_Dist_graph_create()* function. It considers different input parameters:

- old communicator,
- number of source nodes for which this process specifies edges,
- array containing the n source nodes for which this process specifies edges,
- array specifying the number of destinations for each source node in the source node array,
- destination nodes for the source nodes in the source node array,
- weights for source to destination edges,
- hints on optimization and interpretation of weights,
- the process may be reordered (true) or not (false).

It returns a handle to a new communicator to which the distributed graph topology information is attached. As a next step, we call LibTopoMap API, named *TPM_Topomap()*, which

requires the distributed graph communicator, the network topology file, and a variable to return the new id for the MPI rank. LibTopoMap applies different mapping strategies such as *greedy*, *RCM* [9], and *recursive* to optimize the initial mapping. In case an optimization is possible, the MPI ranks get a new id, otherwise nothing changes. With the new ids, we create the reordered communicator, while using the *MPI_comm_split()* function. This functions takes as an input the distributed graph communicator and the new ids for each process to produce a new (*reordered*) communicator. In the end, we perform the same communication of the application to measure the execution time.

5. Results & Discussion

In this chapter, we present the system, the design of the experiments table, and the results of the experiments performed in this work.

5.1 System

All the experiments were executed using miniHPC cluster [3], provider from HPC research group at the University of Basel. The miniHPC has a peak performance of 28.9 double precision TFLOP/s. The miniHPC has two types of nodes, Intel Xeon nodes and Intel Xeon Phi Knights Landing (KNL) nodes. The Intel Xeon nodes amount to 22 computing nodes, 1 login node, and 1 node for storage. The Intel Xeon Phi nodes amount to 4 computing nodes. All nodes are interconnected through two different types of interconnection networks. The first network is an Ethernet network with 10 Gbit/s speed, reserved for users and administrators access. The second network is the fastest network, an Intel Omni-Path network with 100 Gbit/s speed, reserved for the high-speed communication between the computing nodes. The topology of this second network interconnects the 28 nodes (24 Xeons and 4 KNLs) of the miniHPC cluster via a two-level fat-tree topology [3]. It is important to know that during the experiments we used only the Xeon nodes.

5.2 Design of experiments

The design of the experiments is an important process in every research work. In this work, the factorial design of the experiment table consists of different combinations between the mapping strategies, the combination of the number of HCUs and SPEs, and interconnection network topologies. Also, during the experimental phase, different communication evaluation metrics are extracted.

Factors		**Values**	**Properties**
Application		COMAP	A point-to-point application that has the following configurations: a) Method of communication = 1 (blocking) b) Number of exchanges = 100 c) Number of messages = 100. d) Communication pattern = 1 (nearest neighbour (NN))
Mapping strategies	Greedy algorithms	Greedy	Map heavily communicating processes to nodes close to each other recursively.
	Graph partitioning	Recursive bisection	Map based on splitting the minimum weighted edge-cut of the logical and physical communication graph.
	Graph similarity	Reverse Cuthill-McKee (RCM)	Map by turning the adjacency matrices between the physical and logical communication graphs into similar shape.
Network topologies	Indirect	Two-level fat-tree, Dragonfly	Switch-based connection between the computing nodes.
	Direct	3D Torus, 3D Mesh	Direct connection between the computing nodes without having switches.
Computing nodes		miniHPC-Broadwell	Intel Broadwell E5-2640 v4 (2 sockets, 10 cores each) #HCU = 4, 8, 16 #SPUs/HCU = 2, 4, 8, 16
Communication evaluation metrics	Physical communication metrics	Congestion	$Congestion(\Gamma) = max_{uv} Congestion(uv)$
		Dilation	$Dilation(uv) = sum_{u,v \in V_G} R_H(\Gamma_{(u)} \Gamma_{(v)})(p) \cdot \mid p \mid$
	Logical communication metrics	Inter-process logical communication (IePLC)	$weight_i \forall i \in E_{uv}$
		Intra-node logical communication (IaNLC)	$\#SPUs\ per\ HCU$
		Inter-node logical communication (IeNLC)	$IeNLC = \sum_i^N exchanged\ messages$

Table 5.1: Factorial design of experiments.

5.3 Performance analysis

In this section, we present the performance analysis of the experiments. Each subsection contains the results of the experiments executed while using the network topologies defined in Table 5.1.

5.3.1 Experiments using two-level fat-tree network topology

In this subsection, we present the results extracted while executing COMAP, on miniHPC [3] system while using the two-level fat-tree network topology of the system. Measurements for congestion assume the capacity of the links on the physical network topology is 1. The capacity of the links corresponds to network bandwidth, which we assume is not a bottleneck for the communication in this case. Another important aspect of the experiments is the load (balance or imbalance) in communication between processes.

Scenario 1: consists of a load-balanced communication between processes. In terms of communication, it means that all the processes communicate the same volume of information to each other. In Figure 5.1, the measurements for *congestion* are presented.

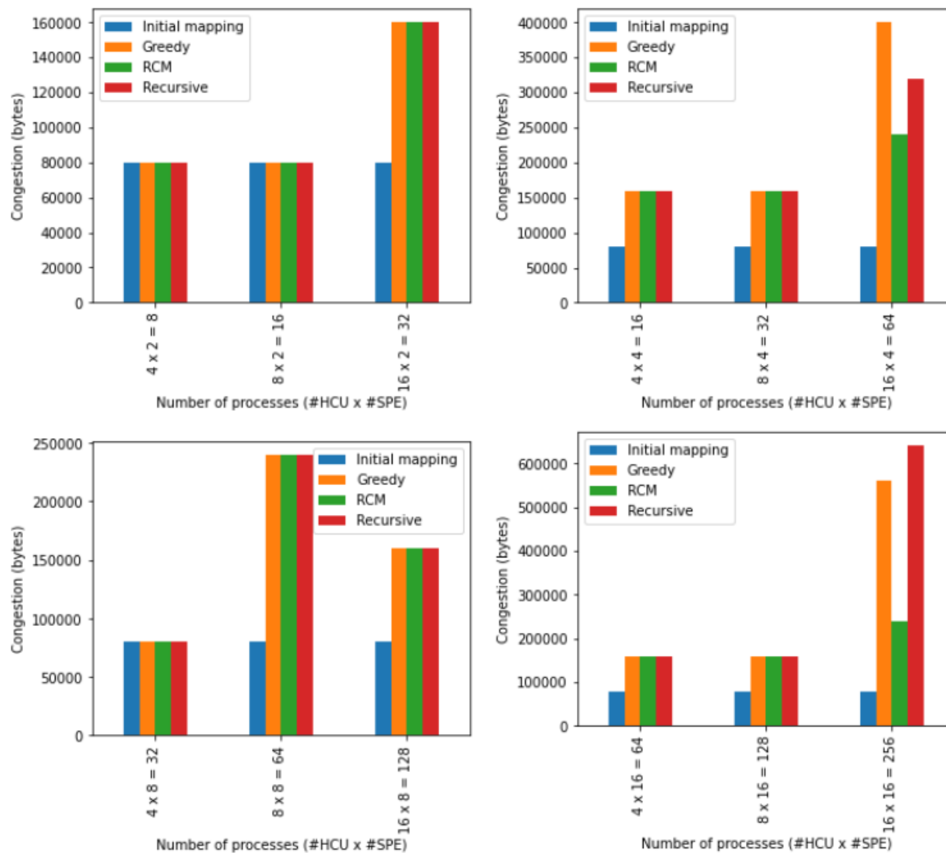

Figure 5.1: Maximum congestion results on different #HCU-#SPE combinations using two-level fat-tree network topology.

Figure 5.1 presents the results that we extracted from the experiments executed in miniHPC with a combination of different numbers of SPEs and HCUs. LibTopoMap calculates the

optimization of processes to nodes placement based on congestion value. Due to this fact and the load-balanced communication between processes, no optimization was found. In some cases, the mapping achieved from the heuristic strategies performs worse than the initial mapping. This is expected behaviour due to the characteristics of the experimentation. The results regarding *dilation* are presented in Figure 5.2.

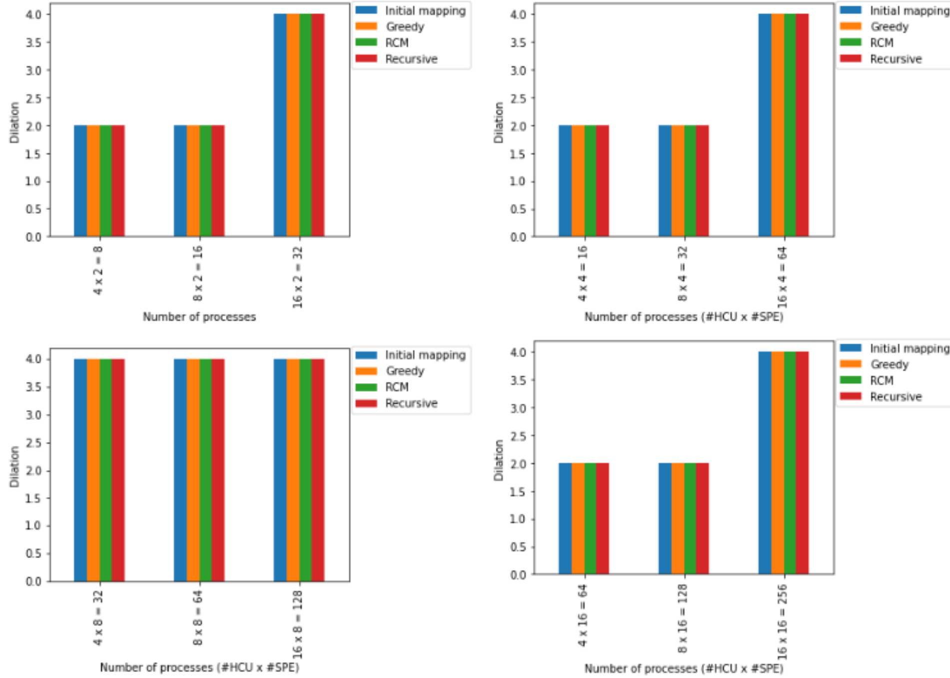

Figure 5.2: Dilation results on different #HCU-#SPE combinations using two-level fat-tree network topology.

Dilation, as shown in Equation 2.1, is the average length of the path taken from a message sent from one SPE to another. We can observe from these results (Figure 5.2) that in all different experiments the mapping strategies perform the same as the initial mapping. A reason for this is that the topology used in these experiments (two-level fat-tree) has a maximum dilation of 4, meaning that from the largest distance between nodes is 4 hops/links. Also, from the results, we can see that when the number of SPEs increases, dilation reaches its maximal value. In it is important to have a further investigation on the dilation values when the total number of SPEs is 32. When the number of HCUs is equal to 8 and the number of SPEs per HCU is equal to 4, dilation is equal to 2. On the other side, when the number of HCUs is equal to 4 and the number of SPEs per HCU is equal to 8, the dilation is equal to 4. That is an example case when one would have to further investigate to find the reason behind these results. In this work, we measure also some logical communication evaluation metrics as shown in Subsection 2.5.2. Therefore, the results regarding logical communication are shown in Figure 5.3.

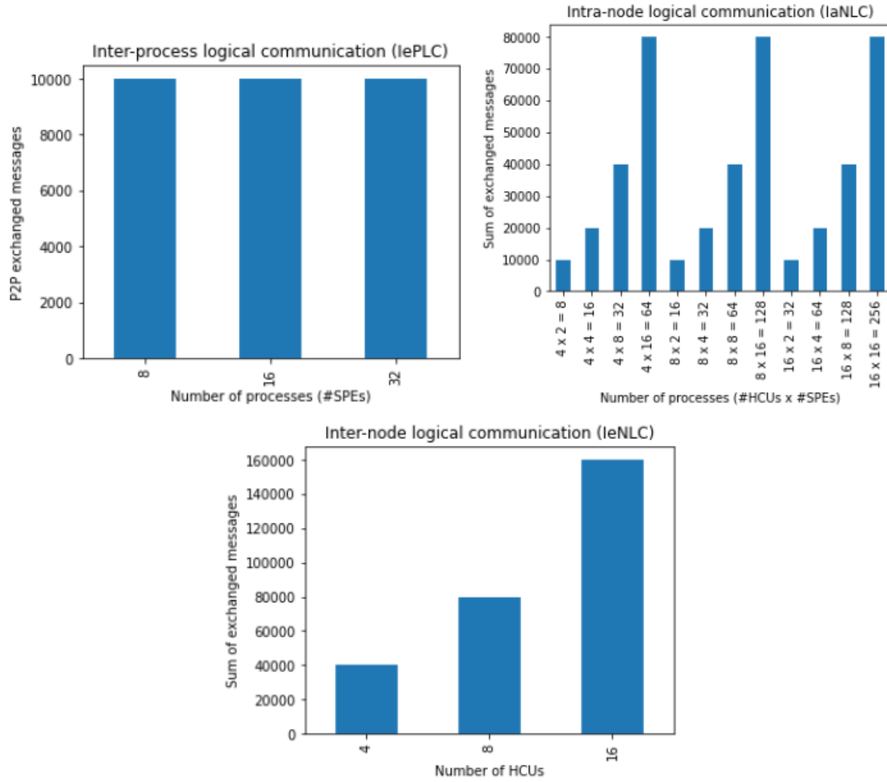

Figure 5.3: Results extracted while using two-level fat-tree network topology.

Inter-process logical communication (IePLC) shows the number of exchanged messages between any pair of SPEs. As we can see from the results, the number of exchanged messages between pairs of SPEs is uniform and all SPEs communicate the same amount of messages. Intra-node logical communication (IaNLC) shows the sum of exchanged messages of SPEs mapped onto the same HCU. It is calculated by multiplying the P2P exchanged messages by the number of SPEs divided by 2. In Figure 5.3, the inter-node logical communication (IeNLC) is presented. It presents the sum of exchanged messages between nodes during execution without considering the physical topology. It is calculated as a multiplication of the number of HCUs with the number of P2P exchanged messages. It is important to mention that during the calculation of the IaNLC, we considered all combinations of HCUs and SPEs. This is because when the number of SPEs differs on different HCUs, the results would be different. During the calculation of IeNLC, we consider that only one process on each node performs inter-node communication, which comes as a result of the NN pattern of communication.

Scenario 2: defines a communication that has a high load imbalance, meaning that each process communicates a different volume of information to another process. As we could witness from the experiments in scenario 1, no optimization of processes to nodes mapping was possible. For this reason, we differentiate the volume of communication sent by each process. We modified COMAP to send vectors of different sizes as the information sent via the MPI communicator. A hash map type of data structure is created, which consists of two variables, *process id* and *the number of elements in the vector.* The rank of the process is

necessary to know which vector will then be transmitted via the MPI communicator. On the other side, the number of elements in the vector for each process is based on the following formula:

$$\text{size of the vector} = 70 + (10 * rank)$$

This formula allows us to have a different volume of information to be sent from each process which therefore would create an imbalance in communication. Another important change is that the number of exchanged messages for each SPE would be a fixed number, which in this case was 30. It is important to know that the number of exchanged messages can be flexibly changed and configured by using the command line arguments. The congestion (in the implementation) is calculated as:

$$\text{congestion} = \text{sizeof(vector[0])} * \text{vector.size()}$$

Congestion is the value that is extracted from multiplying the size (in bytes) of an element in the vector of the SPE and the number of elements in the same vector. We considered the capacity of link 1, in this case as we also did for the executions in scenario 1. Figure 5.4 shows the results extracted from the experiments regarding *congestion*.

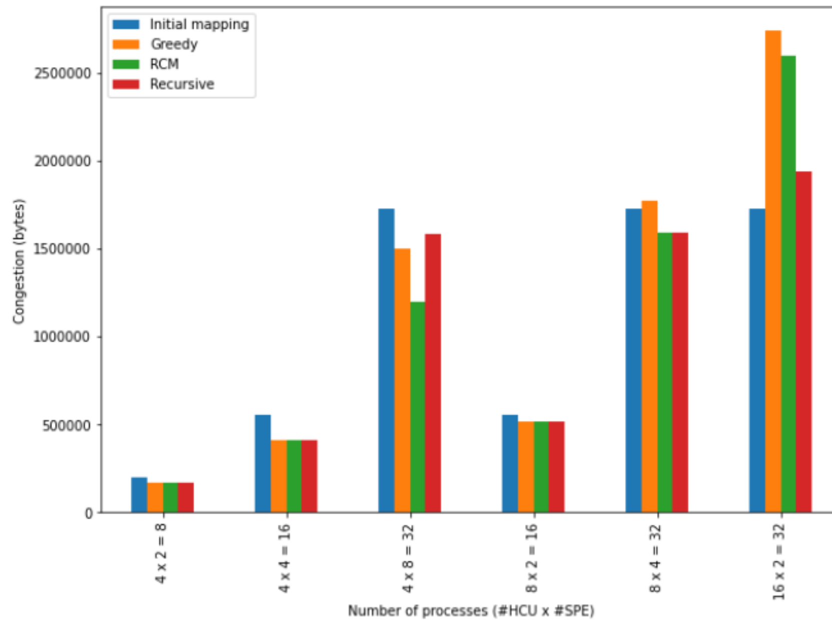

Figure 5.4: Maximum congestion results on different #HCU-#SPE combinations using two-level fat-tree network topology.

In Figure 5.4, we can see the results extracted from the execution with a different number of HCUS and SPEs per HCU. As we can see, due to the way how we calculate the size of the vector for each SPE, the size of the maximum congestion increases proportionally with the number of SPEs used. We can see that when the number of HCUs is small the mapping strategies outperform the initial mapping. As the number of HCUs and therefore also the number SPEs grow, the performance, w.r.t maximum congestion, of the mapping strategies decreases. RCM and Greedy mapping strategies have higher maximum congestion than the

initial mapping when the number of SPEs is equal to 32 (16 HCUs and 2 SPEs per HCU). It is important to know when the value of congestion is higher it means that we could not get a better mapping than the initial one. From the pattern of the results, we can also say that while the number of HCUs, which corresponds to physical units, is small the strategies are able to reduce the congestion. Some of the experiments failed to produce results and therefore are not shown in Figure 5.4. This is due to the limitations of ParMETIS, explained in Subsection 4.1. In Figure 5.5, we present the results extracted from the experiments for *dilation*.

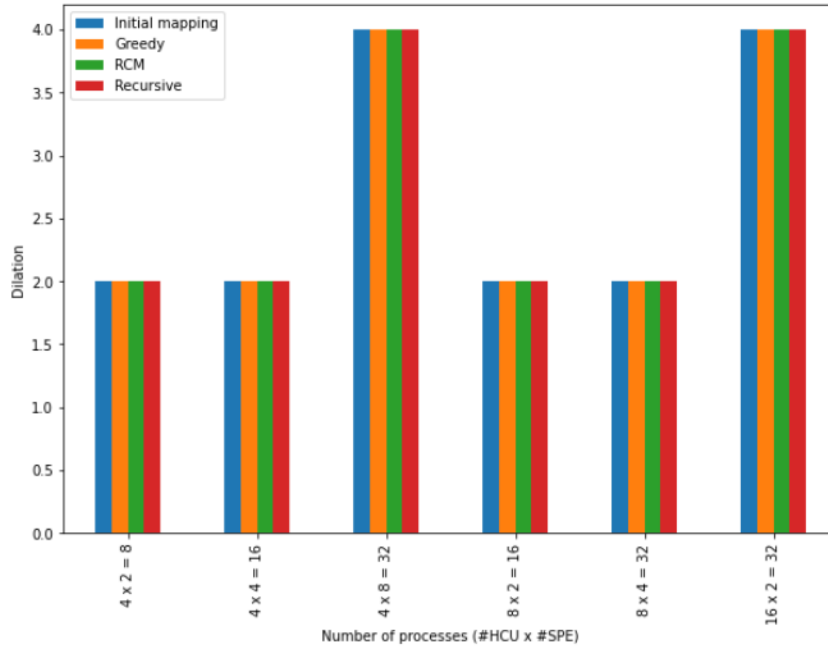

Figure 5.5: Dilation results on different #HCU-#SPE combinations using two-level fat-tree network topology.

We can see that regarding the dilation, there is no change. This is the case, as mentioned in scenario 1 results, because of the interconnection network topology. It is important to know that also in this case the dilation is showing the average length of the path taken from a message in the network topology.

Due to the characteristics of the experiments in this scenario, we measure the communication time of the application with the initial mapping and also with the optimized mappings from the heuristics. These results are presented in Figure 5.6.

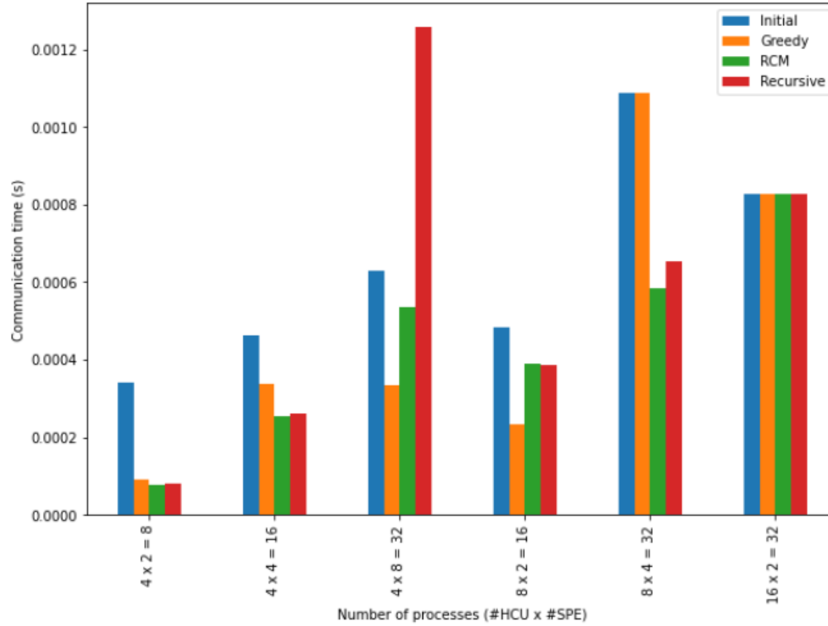

Figure 5.6: Communication time of the application using initial and reordered communicator.

The results in Figure 5.6 consist of communication times of COMAP while using the initial and reordered communicator. The reordered communicator is the one that is being constructed from the reordering of the process ids in LibTopoMap while using the mapping strategies. To extract these results, each execution was repeated 30 times. It is also important to know that an average of initial communication times was taken into account to ensure the correct value (not being disrupted from some other tasks or operating system). On the other side, also an average time was considered from all mapping strategies used in these experiments. From the results, we can see a speedup of more than 50%, in terms of communication time, when the number of SPEs is small. The average speedup that was achieved during the experiments, is around 25%. The communicator taken from the RCM mapping strategy shows a smaller communication time in all the cases. The communicator taken from the Recursive mapping strategy performs well (in terms of communication time), except for the case when 4 HCUs and 8 SPEs per HCU. These results might be still prone to different disruptions coming from the operating system or the network of the HPC system. We can distinguish from these results that not always a reordered communicator performs better than the initial one. The cause of this is not trivial but to be able to understand better, a communication that will take more time to be done is needed.

5.3.2 Experiments using a file describing 3D torus network topology

As mentioned in [18], LibTopoMap allows a simulation mode where the user specifies a mapping of processes to nodes within a file. This allows us to perform experiments on different network topologies other then the system's topology. It is important to know that the experiments in this subsection were performed on a system that uses two-level fat-tree

network topology. LibTopoMap allows to achieve a mapping based on files that describes the network topology of system that might be different from the system where the experiments are performed (explained in Chapter 4). A representation of 3D torus network topology is presented in Figure 2.5, while the configuration file used for these experiments is shown in Figure A.2. The results of the experiments performed while using a file that describes 3D torus network topology are shown in Figure 5.7.

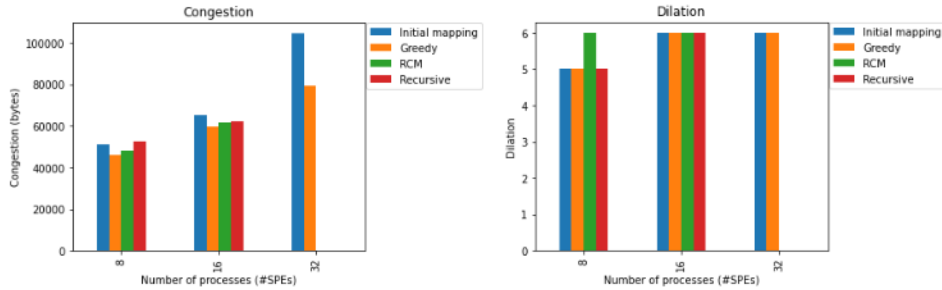

Figure 5.7: Results extracted using a file describing 3D torus network topology (executed on two-level fat-tree topology system).

We can see from the results (Figure 5.7) that we do not differentiate anymore the number of SPEs per HCU, due to the characteristics of the experiments (explained above). From the results, we can derive a limitation of RCM and Recursive mapping strategies. These strategies require the process graph to have lower or the same cardinality as the physical topology graph. Therefore, we can observe that when having more processes (32) than the physical nodes (27, in this case), these two mapping strategies fail to achieve the mapping of processes to nodes. Another interesting aspect, derived from the results, shows that congestion and dilation are not correlated with each other. As we can see the initial mapping performs worse than greedy, RCM, and recursive mapping strategy, in terms of minimum congestion, for 8 and 16 processes, while the dilation is the same. Furthermore, we can see that the dilation metric measured from the RCM mapping strategy is maximal and outperformed all other mapping strategies, including the initial mapping. Furthermore, we expect these experiments to be performed on systems that use 3D torus network topology and extract accurate results. We performed these experiments to test the ability of LibTopoMap to achieve a mapping using arbitrary network topologies and systems.

5.3.3 Experiments using a file describing dragonfly network topology

In this subsection, we present the results that were extracted while executing the experiments using a file describing dragonfly network topology on a system that uses two-level fat-tree network topology (miniHPC). Dragonfly topology is constructed to have 2 groups, 2 chassis, and 5 slots per chassis, as represented in Figure 2.8. The slots are compounded from 4 computing nodes and 1 switch which is used for the routing. All the switches are connected with each other to form the inter-chassis and inter-group connections.

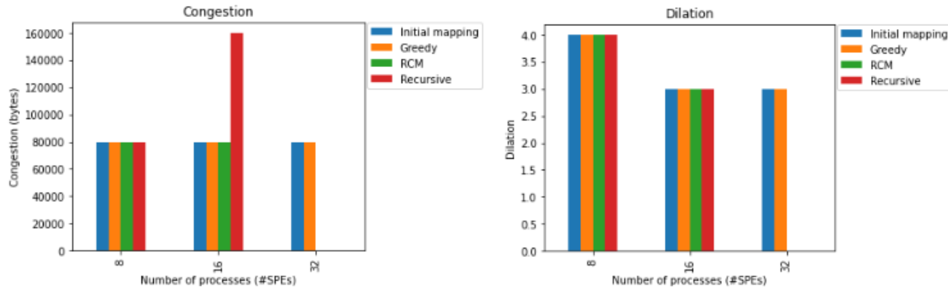

Figure 5.8: Results extracted while using dragonfly network topology

In Figure 5.8, the results were executed while using the dragonfly network topology. We can see similar behaviour of the mapping strategies as in the torus network topology. RCM and Recursive do not achieve any optimization of the mapping due to the constraint of the mapping strategies when COMAP is executed using 32 and 64 SPEs. Initial and greedy mapping show the same results regarding congestion and dilation. This means that we could not get any optimization from the initial mapping done from the operating system. Furthermore, these results comply with the same limitations as the ones mentioned in torus network topology results.

5.3.4 Experiments using a file describing 3D mesh network topology

It is important to know that the experiments in this Subsection were performed on a system that uses a two-level fat-tree network topology. LibTopoMap measures the congestion and dilation values for topologies that are different from the underlying systems' topology. During our experiments with mesh topology, we could not retrieve any results. Due to the nature of the topology and the underlying topology that we used, no mapping of processes to nodes could be possible. These experiments must be performed on a system that uses mesh network topology to be able to do some performance analysis.

6. Conclusion & Future Work(s)

In this chapter, we conclude the work by addressing the limitations, conclusion(s), and future work(s).

6.1 Limitations & challenges

The mapping problem is still a hot topic among researchers but still, there are some limitations and challenges that need to be addressed.

ParMETIS (v3.1.1)[25]: is used in LibTopoMap to partition the process and network topology graphs. During the experiments, we experienced an integer overflow that is triggered from ParMETIS when trying to achieve an edge cut on a weight that is a large number. This error consists of a type (*idxtype*) used in ParMETIS which allows operations until a certain value of integers. As seen in the results section, we were not able to scale up the number of SPEs used during the executions because of this limitation.

Mapping strategies: used in this work are *greedy*, *RCM*, and *Recursive*. We experienced some limitations while taking into consideration different mapping strategies. As we could see from the results section, some of the mapping strategies could not achieve a mapping due to their nature. These strategies require a number of SPEs to be lower or equal to the number of HCUs in the system. To evaluate better the mapping of processes, general heuristics, like greedy, solve the limitations when using multiple SPEs on a HCU on LibTopoMap. A big challenge was raised also when we tried to implement *greedyALLC* and *PaCMap* algorithms. Both these algorithms required the same constraints regarding the number of SPEs and HCUs which made it not possible to retrieve any results using these mapping strategies. To address this challenge, a slight change in the implementation of the algorithms is needed without changing the way how the algorithm works.

System: used in this work is miniHPC. Therefore, we consider having only one system to perform experiments as a limitation. We made use of miniHPC [3], which uses a two-level fat tree network topology. To be able to evaluate different interconnection network topologies, we would have to run experiments on systems that use **3D torus**, *mesh*, and *dragonfly*. Only then we could come up with clearer conclusions.

6.2 Conclusion(s)

In this work, we made use of LibTopoMap [18], a generic library that uses different mapping strategies to optimize the placement of processes to nodes in a HPC system. One of the most important pieces of knowledge extracted during this work is the way how LibTopoMap [18] can be used inside an MPI application so that its optimizations can then be in place to measure the performance gains. We also created COMAP, a configurable MPI commu-

nication application. It can be used in different scenarios since it supports load-balanced or -imbalanced communication. During the experiments, we were able to achieve an average of 20% of speedup on the communication time between SPEs with the reordered communicator from the mapping strategies. Furthermore, we experimented with different network topologies, to show the possible benefits to have the systems running on those strategies.

6.3 Future work(s)

One important aspect would be to address the limitations and challenges that we face during this work. We all are aware of the fact that since the mapping problem is a NP-complete problem, the sub-optimal solutions have a high variety. Therefore, it is difficult to come up with a solution that might satisfy all required aspects. Even though one optimization that can be done to have a broader usage of the LibTopoMap or TopoMatch, is to enhance the workflow used to achieve the mapping. We propose to have the communication matrix of the application and use it in LibTopoMap to achieve an optimization of processes to nodes mapping. In the end, the process ids can be bound to the underlying architecture and the application can be re-executed for performance analysis. With this strategy, one can test the benefits of process binding but on the other side, it can be considered as a limitation of the generality of the solution. This is a trade-off to be considered. Another future work would be to make use of a variety of applications, whose communication is stochastic, to be able to better evaluate the performance gains of the mapping strategies, systems, and interconnection network topologies. In this case, one would have to make changes to the existing libraries to be able to extract the communication and create the necessary inputs that are used to achieve the mapping. Lastly, it is crucial to fix the limitation from ParMETIS, to be able to scale the number of SPEs, used to execute the experiments, up. This could be possibly solved by making LibTopoMap work with the latest versions of METIS and ParMETIS.

Bibliography

[1] Portable Hardware Locality (HwLoc). https://www.open-mpi.org/projects/hwloc/. (Accessed on December 17, 2021).

[2] SuiteSparse matrix collection. https://sparse.tamu.edu/. (Accessed on December 17, 2021).

[3] miniHPC: small but modern HPC. https://hpc.dmi.unibas.ch/en/research/minihpc/. (Accessed on December 17, 2021).

[4] Tarun Agarwal, Amit Sharma, A Laxmikant, and Laxmikant V Kalé. Topology-aware task mapping for reducing communication contention on large parallel machines. In *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*, pages 10–pp. IEEE, 2006.

[5] Daniel Besmer. Process-to-node mapping strategies for the haec box. Master's thesis, University of Basel, 2016.

[6] Shahid H. Bokhari. On the mapping problem. *IEEE Trans. Computers*, 30(3):207–214, 1981.

[7] François Broquedis, Jérôme Clet-Ortega, Stéphanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in hpc applications. In *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, pages 180–186. IEEE, 2010.

[8] Eduardo HM Cruz, Matthias Diener, Laércio L Pilla, and Philippe OA Navaux. An efficient algorithm for communication-based task mapping. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 207–214. IEEE, 2015.

[9] Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172, 1969.

[10] Leonardo Dagum and Ramesh Menon. Openmp: an industry standard api for shared-memory programming. *IEEE computational science and engineering*, 5(1):46–55, 1998.

[11] Mehmet Deveci, Kamer Kaya, Bora Uçar, and Ümit V Çatalyürek. Fast and high quality topology-aware task mapping. In *2015 IEEE International Parallel and Distributed Processing Symposium*, pages 197–206. IEEE, 2015.

[12] Gunter Dueck and Tobias Scheuer. Threshold accepting: A general purpose optimization algorithm appearing superior to simulated annealing. *Journal of computational physics*, 90(1):161–175, 1990.

[13] Gerhard Fettweis, Wolfgang Nagel, and Wolfgang Lehner. Pathways to servers of the future. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1161–1166. IEEE, 2012.

[14] Message P Forum. Mpi: A message-passing interface standard, 1994.

[15] Roland Glantz, Hening Meyerhenke, and Alexander Noe. Algorithms for mapping parallel processes onto grid and torus architectures. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 236–243. IEEE, 2015.

[16] Ananth Grama, Vipin Kumar, Anshul Gupta, and George Karypis. *Introduction to parallel computing*. Pearson Education, 2003.

[17] William Gropp, William D Gropp, Ewing Lusk, Anthony Skjellum, and Argonne Distinguished Fellow Emeritus Ewing Lusk. *Using MPI: portable parallel programming with the message-passing interface*, volume 1. MIT press, 1999.

[18] Torsten Hoefler and Marc Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the international conference on Supercomputing*, pages 75–84, 2011.

[19] Torsten Hoefler, Rolf Rabenseifner, H. Ritzdorf, Bronis R. de Supinski, Rajeev Thakur, and Jesper Larsson Träff. The Scalable Process Topology Interface of MPI 2.2. *Concurrency and Computation: Practice and Experience*, 23(4):293–310, Aug. 2010. ISSN 1532-0634.

[20] Torsten Hoefler, Emmanuel Jeannot, and Guillaume Mercier. An overview of process mapping techniques and algorithms in high-performance computing. *High Performance Computing on Complex Environments*, pages 75–94, 2014.

[21] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. Process placement in multicore clusters: Algorithmic issues and practical techniques. *IEEE Transactions on Parallel and Distributed Systems*, 25(4):993–1002, 2013.

[22] Emmanuel Jeannot, Guillaume Mercier, and François Tessier. Topology and affinity aware hierarchical and distributed load-balancing in charm++. In *2016 First International Workshop on Communication Optimizations in HPC (COMHPC)*, pages 63–72. IEEE, 2016.

[23] Laxmikant V Kale and Sanjeev Krishnan. Charm++ a portable concurrent object oriented system based on c++. In *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, pages 91–108, 1993.

[24] George Karypis and Vipin Kumar. Metis–unstructured graph partitioning and sparse matrix ordering system, version 2.0. 1995.

[25] George Karypis and Vipin Kumar. Multilevelk-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed computing*, 48(1):96–129, 1998.

[26] John Kim, Wiliam J Dally, Steve Scott, and Dennis Abts. Technology-driven, highly-scalable dragonfly topology. In *2008 International Symposium on Computer Architecture*, pages 77–88. IEEE, 2008.

[27] Jonas H Müller Korndörfer, Mario Bielert, Laércio L Pilla, and Florina M Ciorba. Mapping matters: Application process mapping on 3-D processor topologies. *arXiv preprint arXiv:2005.10413*, 2020.

[28] Mohamed F Mokbel, Walid G Aref, and Ibrahim Kamel. Analysis of multi-dimensional space-filling curves. *GeoInformatica*, 7(3):179–209, 2003.

[29] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *International Conference on High-Performance Computing and Networking*, pages 493–498. Springer, 1996.

[30] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel multilevel algorithms for multi-constraint graph partitioning. In *european conference on parallel processing*, pages 296–310. Springer, 2000.

[31] Viacheslav Sharunov. Optimized parallel tasks to nodes mapping in 3-D high performance interconnection topologies. Master's thesis, University of Basel, 2017.

[32] Pirah Noor Soomro, Muhammad Aditya Sasongko, and Didem Unat. Bindme: A thread binding library with advanced mapping algorithms. *Concurrency and Computation: Practice and Experience*, 30(21):e4692, 2018.

[33] Ozan Tuncer, Vitus J Leung, and Ayse K Coskun. Pacmap: Topology mapping of unstructured communication patterns onto non-contiguous allocations. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 37–46, 2015.

[34] Jingjin Wu, Xuanxing Xiong, and Zhiling Lan. Hierarchical task mapping for parallel applications on supercomputers. *The Journal of supercomputing*, 71(5):1776–1802, 2015.

A. Appendix

A.1 Installing and understanding the libraries

In this section, we introduce the steps that are needed to install the libraries: LibTopoMap, TopoMatch, and Maplib.

A.1.1 LibTopoMap

In this subsection, we show the necessary information that could be extracted from this work during the working process with LibTopoMap. It contains information about the installation process, the configuration files, and the library's main API.

A.1.1.1 Installation

To install LibTopoMap in miniHPC, the following steps were performed:

1. Download LibTopoMap package:
   wget https://htor.inf.ethz.ch/research/mpitopo/libtopomap/libtopomap-0.9.tgz

2. Unpack the package:
   tar -xvf libtopomap-0.9.tgz

3. Create a directory for ParMETIS inside libtopomap-0.9:
   mkdir MPIParMETIS

4. Change to that directory:
   cd MPIParMETIS

5. Get ParMETIS v3.1.1:
   wget http://glaros.dtc.umn.edu/gkhome/fetch/sw/parmetis/OLD/ParMetis-3.1.1.tar.gz

6. Unpack ParMETIS:
   tar -xvf ParMetis-3.1.1.tar.gz

7. Get METIS v4.0.1:
   wget http://glaros.dtc.umn.edu/gkhome/fetch/sw/metis/OLD/metis-4.0.1.tar.gz

8. Unpack METIS:
   tar -xvf metis-4.0.1.tar.gz

9. Load the C++ compiler in order to compile ParMETIS

10. Change the directory to ParMETIS:
    cd ParMetis-3.1.1

11. Compile ParMETIS:
    make

12. Change directory to METIS:
    cd metis-4.0

13. Open the file proto.h:
    vi Lib/proto.h

14. Add the diff lines from https://htor.inf.ethz.ch/research/mpitopo/libtopomap/metis_4.0-extern_c-patch.diff to this file. Specifically, when there is + sign in front, those lines should be added.

15. Change the directory to LibTopoMap directory

16. Compile LibTopoMap:
    make

17. Refer to https://htor.inf.ethz.ch/research/mpitopo/libtopomap/ for testing the library.

A.1.1.2 Configuration files

To achieve the mapping of processes-to-nodes, LibTopoMap takes in consideration network topology files. These files can be separated into two categories:

1. HPC system's interconnection network topology
    - The system which one performs the experiments must have its components (nodes) connect using this network topology configuration. This is known as topology-aware mapping.

2. Simulation interconnection network topology.
    - This means that the library will not take into consideration the underlying network topology to execute the experiments, but only the provided network topology. It is known as topology-oblivious mapping.

We performed our executions using miniHPC, which has a network topology structure defined as a two-level fat-tree. in Figure,A.1 we show a network topology file that represents the system's topology that can be interpreted by LibTopoMap.

```
num: 26
0 dmi-cl-node001
1 dmi-cl-node002
2 dmi-cl-node003
3 dmi-cl-node004
4 dmi-cl-node005
5 dmi-cl-node006
6 dmi-cl-node007
7 dmi-cl-node008
8 dmi-cl-node009
9 dmi-cl-node010
10 dmi-cl-node011
11 dmi-cl-node012
12 dmi-cl-node013
13 dmi-cl-node014
14 dmi-cl-node015
15 dmi-cl-node016
16 dmi-cl-node017
17 dmi-cl-node018
18 dmi-cl-node019
19 dmi-cl-node020
20 dmi-cl-node021
21 dmi-cl-node022
22 OmniPathSwitch
23 OmniPathSwitch
24 OmniPathSwitch
25 OmniPathSwitch
0 22
1 22
2 22
3 22
4 22
5 22
6 22
7 22
8 22
9 22
10 23
11 23
12 23
13 23
14 23
15 23
16 23
17 23
18 23
19 23
20 23
21 23
22 0 1 2 3 4 5 6 7 8 9 24 24 24 24 24 24 24 24 24 24 25 25 25 25 25 25 25 25 25 25
23 10 11 12 13 14 15 16 17 18 19 20 21 24 24 24 24 24 24 24 24 24 24 24 24 25 25 25 25 25 25 25 25 25 25 25 25
24 22 22 22 22 22 22 22 22 22 22 23 23 23 23 23 23 23 23 23 23 23 23
25 22 22 22 22 22 22 22 22 22 22 23 23 23 23 23 23 23 23 23 23 23 23
```

Figure A.1: The two-level fat-tree network connection file

The second part of the network topology files that are used in this work consists of different interconnection network topologies such as 3-D torus, 3-D mesh, dragonfly. Since these topologies do not take into consideration the system's underlying network topology, we had to make use of the functionality of LibTopoMap that is known as node naming. We define node naming files that are then used in the network connection file so that the network topology can be used.

```
0 host_0_0_0
1 host_0_0_1
2 host_0_0_2
3 host_0_1_0
4 host_0_1_1
5 host_0_1_2
6 host_0_2_0
7 host_0_2_1
8 host_0_2_2
9 host_1_0_0
10 host_1_0_1
11 host_1_0_2
12 host_1_1_0
13 host_1_1_1
14 host_1_1_2
15 host_1_2_0
16 host_1_2_1
17 host_1_2_2
18 host_2_0_0
19 host_2_0_1
20 host_2_0_2
21 host_2_1_0
22 host_2_1_1
23 host_2_1_2
24 host_2_2_0
25 host_2_2_1
26 host_2_2_2
```

((a)) Node naming file.

```
num:  27
0 host_0_0_0
1 host_0_0_1
2 host_0_0_2
3 host_0_1_0
4 host_0_1_1
5 host_0_1_2
6 host_0_2_0
7 host_0_2_1
8 host_0_2_2
9 host_1_0_0
10 host_1_0_1
11 host_1_0_2
12 host_1_1_0
13 host_1_1_1
14 host_1_1_2
15 host_1_2_0
16 host_1_2_1
17 host_1_2_2
18 host_2_0_0
19 host_2_0_1
20 host_2_0_2
21 host_2_1_0
22 host_2_1_1
23 host_2_1_2
24 host_2_2_0
25 host_2_2_1
26 host_2_2_2
0 9 18 3 6 1 2
1 10 19 4 7 2 0
2 11 20 5 8 0 1
3 12 21 6 0 4 5
4 13 22 7 1 5 3
5 14 23 8 2 3 4
6 15 24 0 3 7 8
7 16 25 1 4 8 6
8 17 26 2 5 6 7
9 18 0 12 15 10 11
10 19 1 13 16 11 9
11 20 2 14 17 9 10
12 21 3 15 9 13 14
13 22 4 16 10 14 12
14 23 5 17 11 12 13
15 24 6 9 12 16 17
16 25 7 10 13 17 15
17 26 8 11 14 15 16
18 0 9 21 24 19 20
19 1 10 22 25 20 18
20 2 11 23 26 18 19
21 3 12 24 18 22 23
22 4 13 25 19 23 21
23 5 14 26 20 21 22
24 6 15 18 21 25 26
25 7 16 19 22 26 24
26 8 17 20 23 24 25
```

((b)) Node mapping file.

Figure A.2: Network connection file for 3x3x3 (3D) Torus topology.

In Figure A.2, the files used to perform the experiments using a 3D torus network topology are shown.

```
0 host_0_0_0
1 host_0_0_1
2 host_0_0_2
3 host_0_1_0
4 host_0_1_1
5 host_0_1_2
6 host_0_2_0
7 host_0_2_1
8 host_0_2_2
9 host_1_0_0
10 host_1_0_1
11 host_1_0_2
12 host_1_1_0
13 host_1_1_1
14 host_1_1_2
15 host_1_2_0
16 host_1_2_1
17 host_1_2_2
18 host_2_0_0
19 host_2_0_1
20 host_2_0_2
21 host_2_1_0
22 host_2_1_1
23 host_2_1_2
24 host_2_2_0
25 host_2_2_1
26 host_2_2_2
```

((a)) Node naming file.

```
num:  27
0 host_0_0_0
1 host_0_0_1
2 host_0_0_2
3 host_0_1_0
4 host_0_1_1
5 host_0_1_2
6 host_0_2_0
7 host_0_2_1
8 host_0_2_2
9 host_1_0_0
10 host_1_0_1
11 host_1_0_2
12 host_1_1_0
13 host_1_1_1
14 host_1_1_2
15 host_1_2_0
16 host_1_2_1
17 host_1_2_2
18 host_2_0_0
19 host_2_0_1
20 host_2_0_2
21 host_2_1_0
22 host_2_1_1
23 host_2_1_2
24 host_2_2_0
25 host_2_2_1
26 host_2_2_2
0 1 3 9
1 0 2 4 10
2 1 5 11
3 0 4 6 12
4 1 3 5 13
5 2 4 14
6 3 7 15
7 4 6 8 16
8 5 7 17
9 0 10 12 18
10 1 9 11 13
11 2 10 14 20
12 3 9 13 15 21
13 4 10 12 14 22
14 5 11 13 17 23
15 6 12 16 24
16 7 13 15 17 25
17 8 14 16 26
18 9 19 21
19 10 18 20 22
20 11 19 23
21 12 18 22 24
22 13 19 21 23 25
23 14 20 22 26
24 15 21 25
25 16 22 24 26
26 17 23 25
```

((b)) Node mapping file.

Figure A.3: Network connection file for 3x3x3 (3D) Mesh topology.

```
0 Node1
1 Node2
2 Node3
3 Node4
4 Node5
5 Node6
6 Node7
7 Node8
8 Node9
9 Node10
10 Node11
11 Node12
12 Node13
13 Node14
14 Node15
15 Node16
16 Switch
17 Switch
18 Switch
19 Switch
```

((a)) Node naming file.

```
num: 20
0 Node1
1 Node2
2 Node3
3 Node4
4 Node5
5 Node6
6 Node7
7 Node8
8 Node9
9 Node10
10 Node11
11 Node12
12 Node13
13 Node14
14 Node15
15 Node16
16 Switch
17 Switch
18 Switch
19 Switch
0 16
1 16
2 16
3 16
4 17
5 17
6 17
7 17
8 18
9 18
10 18
11 18
12 19
13 19
14 19
15 19
16 0 1 2 3 17 17 17 17 18 18 18 18 19 19 19 19
17 4 5 6 7 16 16 16 16 18 18 18 18 19 19 19 19
18 8 9 10 11 16 16 16 16 17 17 17 17 19 19 19 19
19 12 13 14 15 16 16 16 16 17 17 17 17 18 18 18 18
```

((b)) Node mapping file.

Figure A.4: Configuration file for Dragonfly network topology.

A representation of the file used to perform experiments using dragonfly network topology is shown in Figure A.4. This topology has the following characteristics:

- 5 slots which consist of 4 computing nodes and 1 switch,
- 2 chassis per group. Each chassis allocates 5 slots (4 computing nodes and 1 switch),
- 2 groups, where each group has 2 chassis.

A.1.1.3 LibTopoMap API

This subsection contains the most important information about LibTopoMap API. In Figure A.5, we can see the main API that is exposed in LibTopoMap which is used to map processes-to-nodes.

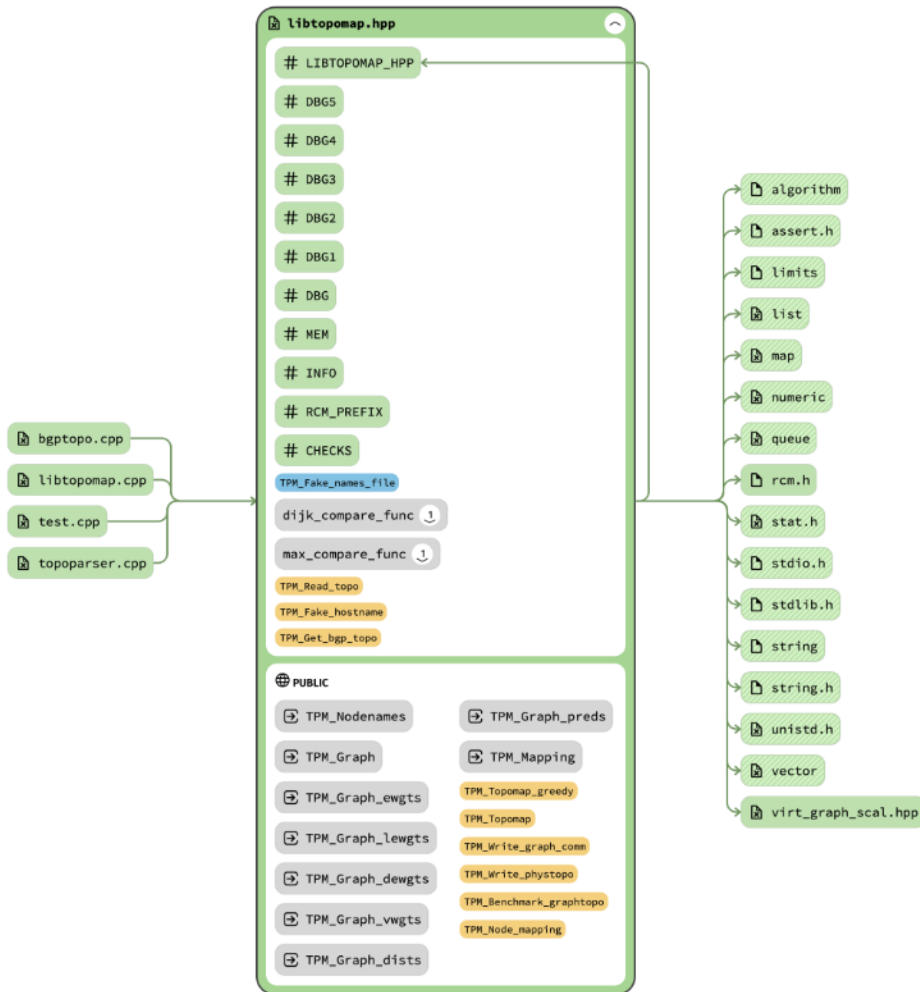

Figure A.5: LibTopoMap API representation

Figure A.5 shows the main files, functions, algorithms, and data structures that are being used in LibTopoMap.

libtopomap.hpp is the file that contains the definition of the functions that are being used in libtopomap.cpp. In this file, we can see that the main function TPM_Mapping() is defined. This function is used to map the logical process to a physical topology graph. We can also see that inside this function, multiple functions are also called. These functions perform operations such as: mapping the graphs by using different mapping strategies, writing the communication graph into output files, benchmarking the time needed for the remapping to happen (in case there is a remapping done). It also contains algorithms that are used to calculate the distances between physical nodes in the physical topology such as Dijkstra Single-Source-Shortest-Path (SSSP) algorithm.

topoparser.cpp is the file that contains the information about the physical and logical

topologies. It checks the physical node names that exist in the system and returns to the main function that uses this information to achieve the mapping.
test.cpp is the file that tests the installation/compilation of the library.

A.1.2 TopoMatch

In this subsection, we show the necessary information that could be extracted from this work during the working process with TopoMatch. It contains information about the installation process and the library's main API.

A.1.2.1 Installation

In order to install TopoMatch in miniHPC, the following steps were performed:

1. Clone the repository of TopoMatch
   git clone https://gitlab.inria.fr/ejeannot/topomatch.git
2. Clone the repository for SCOTCH v6.0
   git clone https://gitlab.inria.fr/scotch/scotch.git
3. Install SCOTCH under my user account by following INSTALL.txt
   note: should consider the ***make prefix=/home/myself/usr/ install***
4. Clone the repository of HwLoc
   git clone https://github.com/open-mpi/hwloc.git
5. Install HwLoc under my user account by following README file (also here the –prefix flag was used to install under my user account)
6. Configure TopoMatch to use SCOTCH
   ./configure SCOTCH_DIR=/dir/to/Scotch/install
7. Compile TopoMatch
   make
8. Perform the tests to check if the compilation went fine
   make check
9. Install TopoMatch under your user
   make –prefix=path/you/want/to_install install
   When no root permission is available, one should install in a path that has access otherwise, permission errors will appear.

After the installation of the library, some example experiments were executed to get some initial insights into how the library works. To execute the example we used the following command:

```
../src/topomatch/mapping -t ./topologies/16.tgt -c ./com_patterns/16.mat
```

where,
../src/topomatch/mapping is the mapping example (that calls the TopoMatch API that perform that mapping with different algorithms). This file calls:

1. tm_load_topology(arch_filename, arch_file_type), that reads the topology defined in a .tgt (tree like topology) or .xml (topology retrieved from HwLoc) file.
2. tm_load_aff_mat(com_filename), that reads the communication pattern of the application. In the example matrices, the cell value of the matrix expresses the affinity between processes.
3. tm_compute_mapping(toplogy, aff_mat, NULL, NULL), that computes the mapping of the communication pattern using TopoMatch algorithm with the given topology.

In Figure A.6, it is shown an image of the process communication pattern (for 16 processes, which represent the affinity between processes) in a) and the tree-like topology in b). I still have to understand how the tree is defined in this case.

```
0 100 19 12 19 11 2511 223 2818 12 10 12589 14 22 177 281
100 0 12 12 25 28 223 1412 199 11 12 281 14 11 2818 15848
19 12 0 223 251 15848 17 10 14 199 112 10 1995 1412 12 10
12 12 223 0 19952 112 17 17 15 2238 1995 17 199 281 14 22
19 25 251 19952 0 158 17 12 17 1584 2238 15 158 141 10 22
11 28 15848 112 158 0 25 19 25 251 158 14 1122 1584 28 12
2511 223 17 17 17 25 0 100 19952 25 28 1258 19 19 158 281
223 1412 10 17 12 19 100 0 177 10 15 251 22 11 22387 1258
2818 199 14 15 17 25 19952 177 0 22 12 1258 19 11 100 199
12 11 199 2238 1584 251 25 10 22 0 11220 17 281 158 28 10
10 12 112 1995 2238 158 28 15 12 11220 0 28 112 223 22 11
12589 281 10 17 15 14 1258 251 1258 17 28 0 11 17 281 223
14 14 1995 199 158 1122 19 22 19 281 112 11 0 22387 19 22
22 11 1412 281 141 1584 19 11 11 158 223 17 22387 0 14 19
177 2818 12 14 10 28 158 22387 100 28 22 281 19 14 0 1778
281 15848 10 22 22 12 281 1258 199 10 11 223 22 19 1778 0
```

((a)) Process communication pattern

```
tleaf 4 2 500 2 100 2 50 2 10
```

((b)) Tree like topology

Figure A.6: Parameters that TopoMatch example considers

The output of the execution is shown Figure A.7. There are three main algorithms used, namely TopoMatch, Packed, and Round-Robin (RR). The output shows the processes (how processes, starting from 0 (left-to-right), are mapped on the nodes).

```
TopoMatch: 0,4,8,12,13,9,2,6,3,14,15,1,10,11,7,5 : 4.86549e+06
Packed: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 : 7.17535e+07
RR: 0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15 : 7.17535e+07
```

Figure A.7: The output of the mapping in TopoMatch

A.1.2.2 TopoMatch API

This subsection contains the most important information about TopoMatch API. In Figure A.8, we can see the main API that is exposed in TopoMatch which is used to map processes-to-nodes.

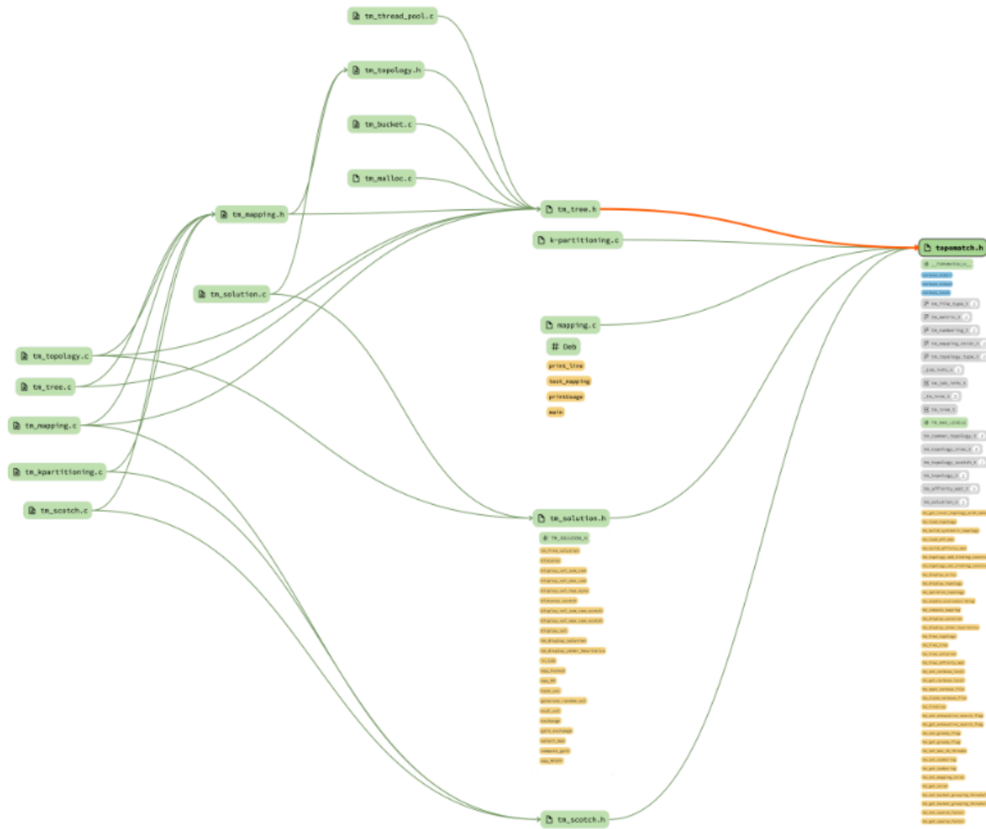

Figure A.8: TopoMatch API representation

mapping.c: in this file, all the directives for the mappings are set, with different types of flags. This file contains a test method that makes use of all the methods in the API that create the mapping. It takes into consideration the architecture file, the process-affinity matrix file and it returns a mapping of the processes to nodes, by using TopoMatch, Round-Robin and Packed algorithm, and information about the metrics that are being measured.
tm_topology.c: in this file different methods to construct the topologies are located. It takes into consideration the topology files that are passed as arguments and it constructs the TopoMatch topology, tree-like topology.
k-partitioning.c: this class is responsible for making use of partitioning the topology, by using scotch. It returns a graph as a topology. It uses different methods to partition the graph.
tm_scotch.c: this class makes it possible to use convert the topology graph to Scotch format. In this way, it can use Scotch to build the mapping, among other strategies.
tm_solution.c: this class is responsible for computing the results for the different metrics used in TopoMatch and also to display the solution.

A.1.3 MapLib

During this work, we enhanced MapLib to support two indirect network topologies, *two-level fat tree* and *dragonfly*. The existing mapping strategies, implemented in MapLib, consider only the direct network topologies. For this reason, the new topologies could not be tested. Even though, one can visualize these strategies with the following command:

```
python3 visualize_topology.py -t dragonfly -s 5 -c 2 -g 2

or

python3 visualize_topology.py -t fattree -p 5
```

The first command is used to visualize dragonfly network topology by defining the topology name (-t), the number of slots (-s), the number of chassis (-c), and the number of groups (-g) in the topology. The second command is used to visualize two-level fat-tree network topology by defining the topology name (-t) and the number of ports for each switch (-p).