UCLouvain

epl

École polytechnique de Louvain

Graph Partitioning and its application to MPI (message passing interface)

Author: **Nathan Tihon**
Supervisor: **Julien Hendrickx**
Readers: **Thomas Gillis, Philippe Chatelain**
Academic year 2023–2024
Master [120] in Mathematical Engineering

Abstract

This thesis lies at the crossroad of graph theory and high performance computing. The rise of increasingly bigger supercomputers has made the distributed computing paradigm more prominent than ever before. The Message Passing Interface (MPI) provides a standard describing how we can take advantage of distributed computing to unleash the full power of these machines. But even the best system in the world is subject to a simple physical limit : the time it takes for data to travel.

In this work, we model distributed computing applications as a graph in which vertices are processes and edges represent the communications in between processes. We then use graph partitioning techniques to group processes on the same node of the supercomputer. In that regard, we attempt to minimize the total communication time of the application. We develop a library for this purpose and apply it to minimize the communication time of FLUPS, a scientific application solving 3D Poisson equations through FFTs. We find that our library is able to reduce the overall runtime of FLUPS by up to 20% on Lucia, a Belgian supercomputer.

We believe the results obtained on FLUPS can be obtained for other type of software such as finite-difference solvers.

Dedication

Before starting this thesis, I would like to give a special thanks to Dr. Thomas Gillis for giving me the opportunity to get out of my comfort zone and live abroad for a couple of months. It has given me the chance to learn a lot about myself and to meet so much incredible people at Argonne. For this reason, I would also like to thank all the Mathematics and Computer Science division at Argonne for making this internship an amazing experience.

For their support during the five years of my studies, I would like to thank my family. They have listened to me explaining courses I found difficult for countless hours even though they did not understand much of it. I am grateful to have them all in my life. Included in the term "family" are Mara and Nathan[1]. They've been there most of my life and continue to do so.

Furthermore, I also thank Alexandre. Despite all our differences, we've been able to support each other during our ups and downs. I hope we will soon celebrate the end of our studies with a final coffee at Paul's.

Finally, I would like to thank my supervisor, Pr. Julien Hendrickx, for his guidance during this thesis.

Acknowledgement

The simulations were performed on the Luxembourg national supercomputer MeluXina. The authors gratefully acknowledge the LuxProvide teams for their expert support.

The present research benefited from computational resources made available on Lucia, the Tier-1 supercomputer of the Walloon Region, infrastructure funded by the Walloon Region under the grant agreement n°1910247.

Disclaimer

Some parts of this thesis have been rephrased by the use of a large language model. The usage of AI in this thesis is strictly limited to the rephrasing of the text. Furthermore, the author has carefully reviewed the text to ensure that it is correct.

[1] Not me, a guy with a unicorn tattooed on his shoulder.

Contents

List of Figures **5**

List of Tables **8**

I Introduction and Preliminaries **9**

1 Introduction **10**
1.1 Context 10
1.2 State of the Art 11
1.3 Goal of the Master's thesis 13

2 Preliminaries **14**
2.1 Graph Theory 14
2.2 High-Performance Computing and the Message Passing Interface 17

II Graph Partitioning and its Application to MPI **21**

3 From Initial Ordering to Graph Representation **23**
3.1 Evaluation of the Time Cost of Single Messages 25
3.1.1 Standard Latency/Bandwidth Model 25
3.1.2 Extended Latency/Bandwidth Model 26
3.2 Aggregating Parallel Edges Having the Same Direction 27
3.3 Aggregating Parallel Edges Having Opposite Directions 28
3.3.1 The Case of a Half-Duplex Network 29
3.3.2 The Case of a Full-Duplex Network 29

4 Partitioning Methods **30**
4.1 The Multi-Level Framework 31
4.1.1 Coarsening 33
4.1.2 Initial Partitioning 37

4.1.3 Uncoarsening 40

5 From Partition to Final Ordering **45**
5.1 Mapping the Partitions 46
5.1.1 Finding the Isomorphism 48
5.1.2 Avoiding the All-Gather Operation 50

6 Results **52**
6.1 Partitioner 52
6.2 Reordering 55
6.2.1 The Systems 55
6.2.2 Proof of concept 56
6.2.3 FLUPS 60

7 Conclusion **66**

A Why Directed Graphs Are Inefficient ? **67**

B OSU Benchmarks [MVA, 2024] **69**

Bibliography **71**

List of Figures

1.1 *Picture of the Aurora supercomputer at Argonne National Laboratory, taken by me. Only the first row of racks are visible, there are 7 more rows hidden behind it. You can see one computing blade equipped with 2 processors and 6 accelerator (Intel GPU) on the bottom left, there are 10624 blades in total on the supercomputer.* . . . 11

2.1 *Example of an edge-weighted graph. The number on each vertex is their unique ID whereas the number on the edges represent the edge's weight. This type of graph will be our main focus.* . . . 15

2.2 *Example of a perfectly balanced partitioning. Perfectly balanced implies that the imbalance factor ε is 0. Consequently, the partitions must have the exact same size. The cut value is the sum of all edges with endpoints in different partitions, here $cut(P) = 6$.* . . . 16

2.3 *Representation of the toy problem : a unidimensional transport equation with second-order finite differences. Here we have chosen to have 20 equidistant data points.* . . . 19

2.4 *First step of HPC application : distribute the problem. Here we distribute the problem shown in Figure 2.3 on 2 ranks. Rank 1 receives the points $\{0, 1, \dots, 9\}$ and rank 2 receives $\{10, 11, \dots, 19\}$* . . . 20

2.5 *Second step of HPC application : communication-computation phase. Here rank 1 sends x_9 to rank 2 and rank 2 sends x_{10} to rank 1. At the end of the communication step, every rank has all the required data to perform a computation step. The communication pattern between the ranks is shown on the bottom of the figure.* . . . 20

2.6 *Control-Flow graph of a complete execution of the rank-reordering. We will start from the user application and how we can generate an edge-weighted graph based on it in Chapter 3. Chapter 4 will then delve into the possible partitioning strategies. Finally, we will explain how we use this partitioning in Chapter 5.* . . . 22

3.1 *Example of a distributed adjacency list. Each rank r knows only its outward neighborhood, represented here by the different colors. The global communication graph is the union of all these local adjacency lists.* 24
3.2 *Evolution of the bandwidth and latency with the size of the messages (in bytes). Values for local and network communication are shown for both the Lucia and MeluXina systems. Exact values can be found in Table B.1 and Table B.2* 26

4.1 *Example of a complete partitioning using the multi-level framework. Note that for clarity, the weights of the vertices is displayed through their size.* 33
4.2 *Representation of the gain bucket structure as defined by Fidduccia and Mattheyses.* 39
4.3 *Visual representation of the gain update. In this chart, v is moved from partition 1 to 2. The gain of its neighbor u is thus updated to reflect the movement.* 40

5.1 *Effect of reordering on data exchange. As processes are not moved, if the data is loaded before reordering, the communications will not result in the intended data exchange.* 46
5.2 *Example of a communication graph with the initial allocation (left). The target allocation is given by the partitioning (right).* 46
5.3 *Partition-preserving isomorphism $f(.)$ between initial and final partitioning. Contrary to Figure 5.2 we have graphically kept the partitions unmoved to fit the reality of the problem.* 47
5.4 *Information available to all processes before/after All-Gather operation.* . . 50

6.1 *Time-to-partition (left) and edgecut (right) of our partitioner with Louvain coarsening ( ) and without coarsening ( ) relative to METIS ( ) for communication graphs obtained from FLUPS.* 53
6.2 *Time-to-partition and edgecut of our partitioner with Louvain coarsening ( ) and without coarsening ( ) relative to METIS ( ) for theoretical 2D grids. The number above METIS bars represents the fraction of samples respecting the perfect balance constraint.* 54
6.3 *Representation of the proof-of-concept user application with $N = 8$ ranks. (Left) Suboptimal initial linear allocation. (Right) Final reordered allocation.* 57
6.4 *Time-to-solution of the example user app described by Figure 6.3. We display the optimal allocation ( ), with no network communications. Linear allocation ( ) with every communication occurring on network. Ours ( ) is the linear allocation to which we applied our rank reordering algorithm. For clarity, we added the value of the linear allocation above its bars.* . . 58

6.5 *Time-to-solution of the **modified** example user app. We display the optimal allocation ( ) with no network communications. Linear allocation ( ) with every communication occurring on network. Ours ( ) is the linear allocation to which we applied our rank reordering algorithm. For clarity, we added the value of the linear allocation above its bars.* 59
6.6 *Example of FLUPS's topology switches in a 2D case with 16 ranks. The solid lines represent the data local to each rank whereas the dashed lines represent the chunks of data that need to be exchanged during the topology switch.* 61
6.7 *Time-to-solution of FLUPS with and without reordering on up to 128 ranks for both Lucia and MeluXina. We use* 64^3 *data points per rank in a fully unbounded domain. Cross-hashed bars are the FFTs, hashed bars represent the time spent computing after initiating the exchange requests, plain bars represent waiting time. The shades are the different topology switch.* 63
6.8 *Time-to-solution of FLUPS with and without reordering on up to 128 ranks for both Lucia and MeluXina. We use* 96^3 *data points per rank in a fully unbounded domain. Cross-hashed bars are the FFTs, hashed bars represent the time spent computing after initiating the exchange requests, plain bars represent waiting time. The shades are the different topology switch.* 64

List of Tables

5.1 *Values of the pair $(P(r), R(r))$ described above for each rank and for both the initial and final partitioning of the example in Figure 5.3* 49
5.2 *Update of Table 5.1 including the inverse of the partition-preserving isomorphism.* 51

6.1 *Cluster-node properties of both MeluXina and Lucia supercomputers. Note that memory is given in giga-BYTE (GB) and bandwidth is expressed in giga-BIT (Gb), therefore the theoretical bandwidth are 25GB/s and 12.5GB/s.* 55
6.2 *Domain decomposition with regard to the number of cluster-nodes. We also show the successive decompositions for each topology switch.* 62

B.1 *Results of the OSU-benchmarks `osu_bw` and `osu_lat` on network communication for both MeluXina and Lucia systems. Message size is expressed in bytes, latency in μs and bandwidth in MB/s.* 69
B.2 *Results of the OSU-benchmarks `osu_bw` and `osu_lat` on local communication for both MeluXina and Lucia systems. Message size is expressed in bytes, latency in μs and bandwidth in MB/s.* 70

Part I

Introduction and Preliminaries

Chapter 1

Introduction

1.1 Context

The ever-growing demand for computational power has fueled the development of massive supercomputers such as Lumi, Frontier and Aurora [Strohmaier et al., 2023]. These machines interconnect processing units such that they can communicate data with each other. Scientists and researchers leverage these supercomputers to tackle large scale problems in fields such as computational fluid dynamics, nuclear physics, neuroscience modelling and much more. However, maximizing the efficiency of these colossi becomes increasingly crucial as their size grows.

While shared memory parallelism models like OpenMP provide a convenient approach for parallel computing on single machines, scaling them to distributed systems proves challenging and can hinder the performances of even smaller applications [Hu et al., 1999]. This is why the Message Passing Interface (MPI), implemented by libraries like MPICH and OpenMPI, has become the industry standard for utilizing supercomputers. Contrary to shared memory models, MPI operates using processes with independent memory space. It defines an interface for exchanging data between these processes, regardless of their physical location on the machine.

The sheer scale of these supercomputers introduces a critical factor: *communication time*. To grasp the vastness of these systems, consider Argonne National Laboratory's own supercomputer : Aurora (Figure 1.1). This giant machine occupies 930 square meters of space. Since data travels at finite speed, communication between processors close together is naturally faster than across large distances within the machine[1]. This simple observation underlines the impact of process placement on communication speed within distributed memory computers. **Strategic placement can significantly improve communication speed.**

[1]This has been leveraged in the Apple M1 chip using *unified* memory, i.e. memory welded directly on the processor chip.

Our mission is to provide a user-friendly library designed to minimize the communication time of distributed applications running on supercomputers. We achieve this by optimizing the placement of processing units on the machine.

Figure 1.1: *Picture of the Aurora supercomputer at Argonne National Laboratory, taken by me. Only the first row of racks are visible, there are 7 more rows hidden behind it. You can see one computing blade equipped with 2 processors and 6 accelerator (Intel GPU) on the bottom left, there are 10624 blades in total on the supercomputer.*

1.2 State of the Art

Minimizing communication time has been a recurrent problem in the computational engineering community since the speed of the central processing unit (CPU) exceeded the speed of data transfer. In 1980, an Intel 8080 had a 1MHz clock speed and accessed memory once every microsecond, at this time the memory access speed was 300 nanoseconds. Increasing CPU clock speed was more important. Since then, CPU have become much faster than memory.

To lower access time to data, the solution was to develop better memory hardware and cache hierarchy [Carvalho, 2002]. This provided a huge benefit for all areas of computing. Still, the access time to distant memory in distributed applications led to performance loss. In many cases, these accesses can either be reduced by better load balancing or sped up with better data placement. Our focus is on the latter.

To find better data placements, we model communications using a graph. Each vertex of the graph represents a process of the distributed application and the edges the communications between processes. We then use graph partitioning to group heavy-communicating

processes together and place them close on the underlying hardware, this task is called *rank-reordering* and the reason it is named that way will be clear in the last chapter.

The partitioning problem is known to be NP-hard. Along the years, several heuristics have been proposed for this task.

Recursive Bisection is the generalization of Nested Dissection [George, 1973] for arbitrary graphs. It has been used in a large range of state-of-the-art partitioners. This technique recursively partitions the graph into two subgraphs of approximately equal size by minimizing the number of edges cut. This heuristic suffers from the look-ahead bias, a common problem among greedy algorithms.

Spectral Partitioning [Hendrickson and Leland, 1995a] uses the eigenvectors of the Laplacian matrix of the graph to retrieve a partitioning. This technique has the benefit of using global information to compute the partitioning. The downside is the time-consuming computation of eigenvectors hence rendering this technique impractical for large graphs.

Graph Growing is a family of heuristics starting with partitions containing a single vertex and adding vertices adjacent to the partitions at each step, for example using a breadth-first exploration. An example of such technique can be found in [Predari and Esnard, 2016].

Streamed Partitioning is a novel family of heuristics, also called "1-pass partitioning" due to the fact that it requires a single pass on the graph to produce a partitioning. These heuristics are useful if the graphs cannot fit in memory, but generally produce worse partitioning as they only rely on limited local information about the graph. [Abbas et al., 2018]

Several other heuristics have been introduced to improve the quality of existing partitions. The most famous heuristics are the Kernighan-Lin (KL) [Kernighan and Lin, 1970] algorithms, running in quadratic time and operating between pairs of partitions to find groups of vertices to swap between them.

The Fiduccia-Mattheyses (FM) [Fiduccia and Mattheyses, 1982] is a more efficient heuristic, running in linear time. It uses an efficient data structure to update critical information about the movement of vertices between partitions.

Both KL and FM heuristics are only able to move vertices between a pair of partitions. More recent heuristic based on negative cycles in the quotient graph of the partitioning [Sanders and Schulz, 2012] are able to perform non-trivial vertex movements between an arbitrary number of partitions.

Multilevel partitioning scheme [Hendrickson and Leland, 1995b] is an important advance in graph partitioning due to its ability to partition graph of much greater scale. Multilevel

partitioning consists of three phases. In the first phase, the graph is contracted iteratively to form a hierarchy of smaller graphs. During the second phase, the smallest graph of the hierarchy is partitioned using a chosen partitioning algorithm. Finally, the third phase expands the graph and propagates the solution to finer graphs while performing local optimizations using for example the FM algorithm. This technique is used by the state-of-the-art partitioner METIS [Karypis and Kumar, 1999] and KaHIP [Schulz et al., 2019, Sanders and Schulz, 2012], among others.

1.3 Goal of the Master's thesis

Despite the advances in graph partitioning and its potential usage in distributed computing, MPICH does not yet provide the ability to reorder the ranks of an MPI application using the aforementioned techniques. We propose to develop a user-friendly library performing the rank reordering task. Futhermore, this library stays close to MPICH's API such that it can easily be integrated into existing applications.

The contributions of this Master's thesis are thus twofold.

Reordering: On one hand, we develop a reordering library – written in the C language – the goal of which is to compute a rank-reordering. We detail how this library is able to create the communication graph of the processes, and how it reorders the processes using a partitioner. Furthermore, this library is highly modular and allows the usage of multiple partitioners through simple options, making it a very powerful tool to compare the effects of different partitioners on a given HPC application. The library is available at the following URL :

`https://github.com/MonkD3/mpi-rr`

Partitioning: On the second hand, we explore different partitioning strategies and develop a partitioner tailored to the reordering of the ranks of the distributed software FLUPS [Caprace et al., 2021, Balty et al., 2023] which efficiently solves Poisson equations using Fast Fourier Transforms.

Finally, we use both of the above tools to compare the time-to-solution of FLUPS in the following scenarios :

- Without reordering (reference case).
- With reordering issued from our tailored partitioner.

Chapter 2

Preliminaries

The subject of this thesis is at the intersection of both Graph Theory and Distributed Computing. To make sure both mathematicians and computational engineers understands the concepts discussed throughout these pages, we provide a small reminder of the important concepts of both fields.

The reader familiar with both subjects can jump to Chapter 3.

2.1 Graph Theory

In this section, we recall the essentials graph theory concepts. Starting from the fundamentals, a *graph* G is a couple $G = (V, E)$ of sets where :

- V is a set of vertices. In the literature, they are often referred to as *nodes* due to their heavy usage in computer science and electrical engineering. Note that we will strictly use the term *vertex* in this text when discussing graphs. The term *node* introduced later holds a different signification.

- $E \subseteq V \times V$ is a set of edges, also called arcs or links. These edges may be directed or undirected. In the case of undirected edges we will always have $(u, v) \in E \Leftrightarrow (v, u) \in E$.

A graph can be used as a tool to model a great range of topics, for example in electrical and computer engineering, an electrical circuit can be modelled as a graph where the components are the nodes and the circuits are the edges. As explained in the introduction, we use graphs to model the communication patterns of distributed high-performance computing applications.

We extend this definition to weighted graphs. A *weighted graph* is a graph associated with at least one of the following weight functions :

- $c : V \mapsto \mathbb{R}_+ : v \mapsto c(v)$ which associates a weight to each vertex of the graph.
- $w : E \mapsto \mathbb{R} : e \mapsto w(e)$ which associates a weight to the edges of the graph.

The type of weight functions will be clear from the context. In the case of unweighted graphs, we assume a unit weight for both vertices and edges. Generally, we use the shorthand notation w_{ij} for the weight of the edge linking the vertices i and j and c_v for the weight of the vertex v. Figure 2.1 displays a simple example of an edge-weighted graph.

This type of graph can be used whenever we need to encode additional data on the graph. In our use-case, we add weights to the edges to represent the time it takes for two processes to communicate.

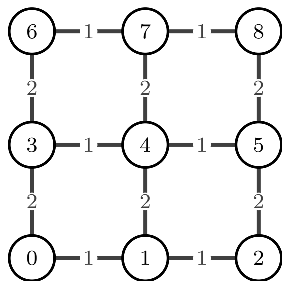


Figure 2.1: *Example of an edge-weighted graph. The number on each vertex is their unique ID whereas the number on the edges represent the edge's weight. This type of graph will be our main focus.*

To ease notations, we extend the weight functions to sets in the following manner :

$$\forall\, V' \subseteq V \;:\; c(V') = \sum_{v \in V'} c(v) \qquad \forall\, E' \subseteq E \;:\; w(E') = \sum_{e \in E'} w(e)$$

This definition simply means that the weight of a set is equal to the sum of the weights of its parts.

We will now talk about the problem of dividing a graph into disjoint subgraphs. Historically, we have been interested in dividing a graph into k parts such that the sum of the edge's weight with endpoint in different parts is minimized. This is the k-way partitioning problem. As we will see later, it appears a lot in distributed computing. More recently, we have been interested in dividing a graph into k disjoint parts such that the nodes in a group are more similar to those in its group than to those in other groups. This is called the k-way clustering problem and is useful to extract the community structure of a social network. Let us now define those two problems more rigorously.

A *k-way partitioning* of a graph $G = (V, E)$ is a set $P = \{P_1, P_2, \ldots, P_k\}$ such that :

- P is a partitioning of the vertices:

$$\begin{cases} \text{All vertices are in a partition} & \bigcup_{i=1}^{k} P_i = V \\ \text{The vertices are only in 1 partition} & \bigcap_{i=1}^{k} P_i = \emptyset \end{cases} \tag{2.1}$$

- There is a balance constraint on the size of the partitions, with an acceptable *imbalance* given by the <u>*imbalance factor*</u> $\varepsilon \geq 0$:

$$c(P_i) \leq \frac{c(V)}{k}(1+\varepsilon) \qquad i = 1, 2, \ldots, k$$

To be clear, P_i denotes a set of vertices, and we will write $P(u) = i$ if $u \in P_i$, meaning that the vertex u is in the i-th partition.

To compute the quality of a partitioning, we use the <u>*edgecut*</u> metric. Let P be a partitioning of the edge-weighted graph $G = (V, E)$, the edgecut metric is then defined as the sum of the weights of the edges with endpoints in different partitions:

$$cut(P) = \frac{1}{2} \sum_{v \in V} \sum_{u: P(u) \neq P(v)} w_{uv} \tag{2.2}$$

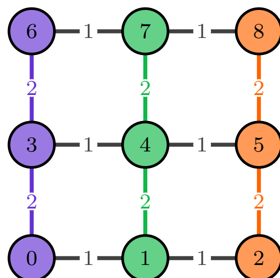


Figure 2.2: *Example of a perfectly balanced partitioning. Perfectly balanced implies that the imbalance factor ε is 0. Consequently, the partitions must have the exact same size. The cut value is the sum of all edges with endpoints in different partitions, here $cut(P) = 6$.*

A <u>*k-way clustering*</u> of a graph $G = (V, E)$ is similar to a k-way partitioning except that we do not enforce a balance constraint. The only constraint for this problem is (2.1). Notice that using the edgecut metric (2.2) without constraint on the partition's size lead to a trivial problem, it suffices to put every vertex into a single partition to have an edgecut of 0. Therefore, k-way clustering uses a different objective based on the similarity of vertices

belonging to the same cluster. A common choice of metric is the modularity Q described for example in [Newman, 2006].
There exists efficient algorithms for the modularity maximization problem, famous examples include the Louvain [Blondel et al., 2008] and Leiden [Traag et al., 2019] algorithms.

Although partitioning and clustering have a similar mathematical formulation, it is important to remark that they are used for different purposes :

- Partitioning is used to minimize interactions between partitions while enforcing balance. We can think about it as a load balancing task.
- Clustering is used for structural analysis of large scale graphs, in which we try to separate clusters that share similar properties. We can think about it as a "community" detection task.

2.2 High-Performance Computing and the Message Passing Interface

In this section, we recall the structure of a supercomputer and how High-Performance Computing (HPC) applications use them efficiently.

A *supercomputer* is a computing platform hierarchically organized as follows:

1. **Thread** : A thread is the sequence of instructions of a program. A thread can either be *running* meaning that the instructions of the program are being executed, *asleep* meaning that the program is not ready to be executed (e.g. waiting for user input, reading from file) and *ready*, meaning that it is ready but not yet executing.
2. **Core** : A core is the hardware capable of executing a thread. Newer cores equipped with the *Hyperthreading* technology can execute 2 threads concurrently. A classical consumer desktop processors generally has 8-16 cores whereas supercomputer and datacenter processors have 32-64.
3. **Socket** : A socket corresponds to a "slot" on a motherboard.
4. **Node** : A node is a group of socket sharing a fast network connection. In this text, we will often refer to resources being *local*, meaning it is located on the same node or *distant*, meaning located in a different node. We also sometimes refer to nodes as *cluster-nodes* to avoid confusion with the graph's vertices.

For example, the system MeluXina has 573 standard nodes. Each node has 2 sockets with 64 cores and each core can execute 2 threads at once. At full scale, the system thus has 256 cores per node which leads to a little less than 147.000 computing units.

An important remark is that a core can actually execute an arbitrary amount of threads, but it cannot do so concurrently. This has important consequences in HPC applications for the simple following reason : a thread currently executed (in *running* state) may require results of threads that are not running (in *ready, asleep* states). Causing it to wait indefinitely and lose time. For this reason, HPC applications will restrict themselves to the maximum number of threads concurrently executable. For example 147.000 on MeluXina.

The memory of supercomputers is also organized hierarchically. The lower levels of the hierarchy corresponds to the processor cache and is akin to consumer desktops. We will not explain them in detail here because they are not of direct importance to our application. The reader interested in leveraging cache hierarchy for performance maximization can consult [Drepper, 2007], which still applies to this day.

The higher levels of the hierarchy corresponds to the main memory. Nowadays, a consumer desktop will be equipped with 8 to 16 Go of Random-Access Memory (RAM). Supercomputers use the same technology in much greater scale, ranging from 256Go to multiple terabytes of memory per node. To have better control over the local and distant memory, supercomputers introduce additional levels in the memory hierarchy.

1. Non-Uniform Memory Access Domain (NUMA Domain) : Each node of a supercomputer is divided in multiple NUMA domain. Each core of the node is assigned to one NUMA domain and can efficiently access memory located inside its NUMA domain. A core is able to access memory of other NUMA domain with increased access time.

2. Network access : if a core needs to access memory located on a different node, a network communication is required. As the core is not located on the same hardware, the data has to be transferred through the network before being accessible.

This brings us to the next important topic : the usage of the *Message Passing Interface* (MPI) [Message Passing Interface Forum, 2023]. When using MPI, a user will start multiple processes that will exchange data between one another. As opposed to classical shared memory (or *multithreaded*) application, in which every thread can access all the memory of the program. Here, each process has its own memory that is not accessible by the other processes. The MPI standard defines ways of communicating data between processes.

Before describing an example use case of scientific MPI application, let us remind some terminology :

- A *message* is a communication of data between two processes. The communication can be unidirectional or bidirectional.

- A *communicator* is a group of processes able to exchange messages. The initial communicator is always `MPI_COMM_WORLD` and corresponds to all the processes of the application. Communicators are useful to define *collective operations*, which are communications involving more than two processes.

- The *rank* is the ID of a process in a communicator. **It is critical to understand that a process can have different ranks on different communicators**. When speaking about a given communicator we generally use "process" and "rank" interchangeably.

In general, an HPC application will evenly distribute the computational load between ranks (a problem that can also be solved using graph partitioning!) and alternate between computation and communication phase. In the computation phase, little to no messages are sent whereas in the communication phase no computations occur. Note that there are more sophisticated techniques than the computation-communication cycle (e.g. latency hiding) that we will not get into. We illustrate this with a simple example : a one dimensional, second-order finite difference scheme for solving a constant velocity transport equation. We then distribute this problem on 2 ranks. This example takes the form :

$$\frac{\partial u}{\partial t} + c\frac{\partial u}{\partial x} = 0 \qquad u(0,x) = f(x) \qquad u(t,0) = u(t,1) = g(t)$$

Let us discretize the domain in n parts of length h and denote $x_j = jh, j = 0, 1, \dots, n$. The spatial discretization of the equation is given by :

$$\frac{\partial u(x_j,t)}{\partial t} + c\frac{u(x_{j+1},t) - u(x_{j-1},t)}{2h} = 0 \qquad \forall j = 0, 1, \dots, n \tag{2.3}$$

It is important to see that we need both neighboring values x_{j-1} and x_{j+1} to evaluate the time derivative at the point x_j. A visual representation of the problem and the discretization is given by Figure 2.3.

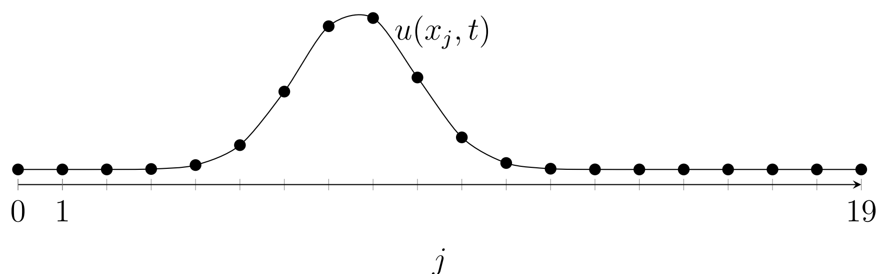


Figure 2.3: *Representation of the toy problem : a unidimensional transport equation with second-order finite differences. Here we have chosen to have 20 equidistant data points.*

When distributing the problem on the 2 ranks, one can easily see that the optimal distribution is to place 10 data points on the first rank and 10 on the other. Note that this distribution becomes difficult in higher dimensions. The distribution is shown in Figure 2.4.

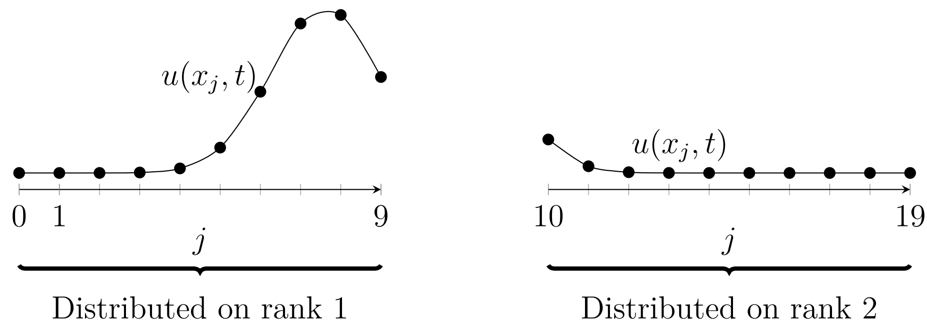


Figure 2.4: *First step of HPC application : distribute the problem. Here we distribute the problem shown in Figure 2.3 on 2 ranks. Rank 1 receives the points* $\{0, 1, \ldots, 9\}$ *and rank 2 receives* $\{10, 11, \ldots, 19\}$

Each rank has now its own local domain. One problem remains : it is not currently possible to evaluate (2.3) at the boundary points x_9 and x_{10} because rank 1 does not have access to x_{10} and conversely rank 2 cannot access x_9. This is why communication takes place. In each iteration, rank 1 will send the value of x_9 to rank 2 and rank 2 will send the value of x_{10} to rank 1. After this exchange is complete, each rank can process its domain without any additional communication. This is represented in Figure 2.5

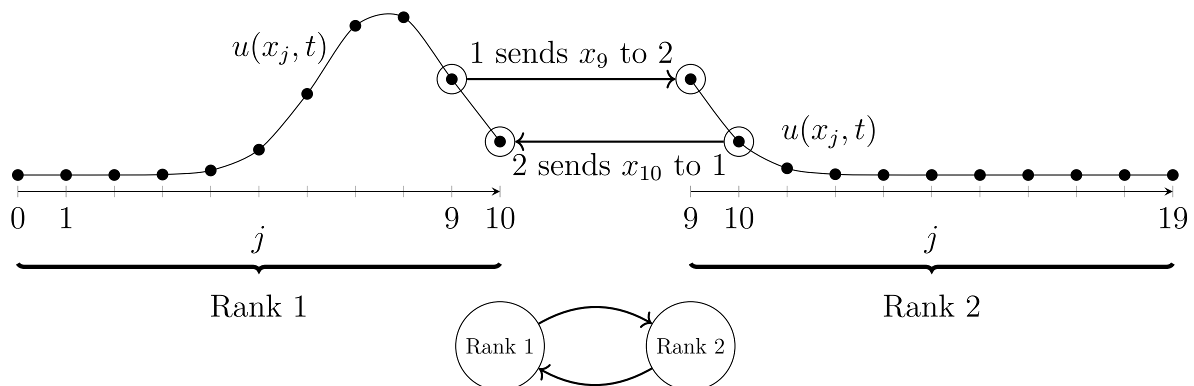


Figure 2.5: *Second step of HPC application : communication-computation phase. Here rank 1 sends* x_9 *to rank 2 and rank 2 sends* x_{10} *to rank 1. At the end of the communication step, every rank has all the required data to perform a computation step. The communication pattern between the ranks is shown on the bottom of the figure.*

Part II

Graph Partitioning and its Application to MPI

In this part, the ordering of the text closely follows the execution flow of our underlying libraries. In this way, the reader can get a good idea of the steps undertaken by the solution simply by looking at the table of contents.

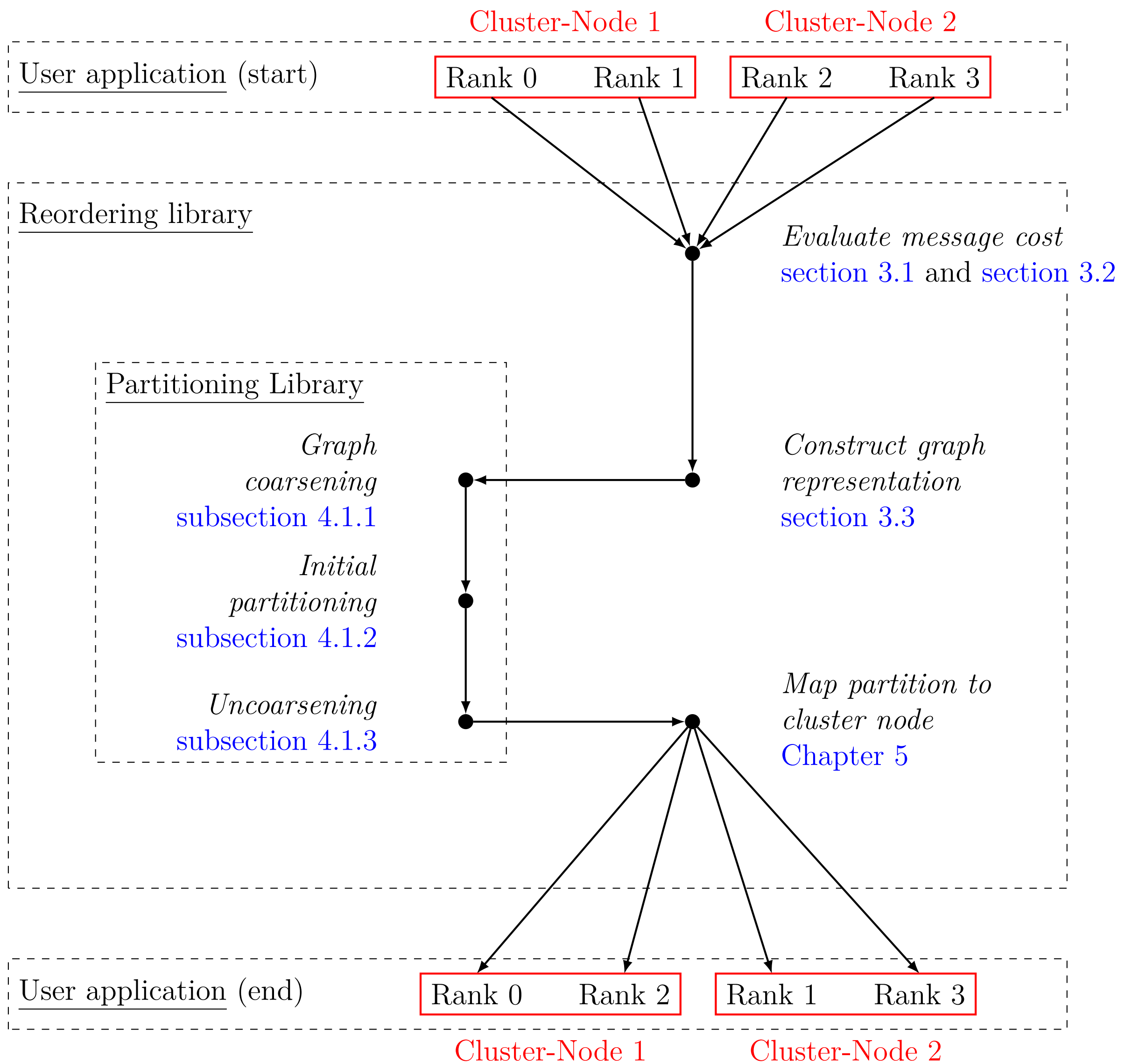


Figure 2.6: *Control-Flow graph of a complete execution of the rank-reordering. We will start from the user application and how we can generate an edge-weighted graph based on it in Chapter 3. Chapter 4 will then delve into the possible partitioning strategies. Finally, we will explain how we use this partitioning in Chapter 5.*

Chapter 3

From Initial Ordering to Graph Representation

In this chapter, we introduce the first half of our reordering library, responsible to create the communication graph. This is a non-trivial step due to the distributed nature of the application. Each rank knows the number, size and destination of the messages it sends, but it does not have the same information regarding other ranks. In graph theoretic terms, this means each rank carries the information of its direct outward neighborhood. We thus require a synchronization phase to gather the entire topology of the graph to be able to feed it to the partitioner.

The second half of the library will be discussed in Chapter 5.

By definition of an MPI program, the communications are directly programmed by the user. Consequently, the user knows exactly the sender and receiver of each message as well as the size of these communications. We ask that the user provides the distributed list of messages of each rank to the reordering library. In other words, each rank will provide to the library a list of all the messages it sends. **As such, the reordering library does not require any additional user-provided information about the application**. Furthermore, later release of this library might allow the user to first run a short version of their application while recording all communications with for example pilgrim [Wang et al., 2023]. Then using this data to construct the communication graph[1].

Assuming we have N ranks. Each rank $r \in \{0, 1, ..., N-1\}$ sends m_r messages of size w_i towards the target t_i, $i \in \{1, 2, ..., m_r\}$. We are given N lists of the form $[(t_i, w_i)]_{i=1}^{m_r}$.

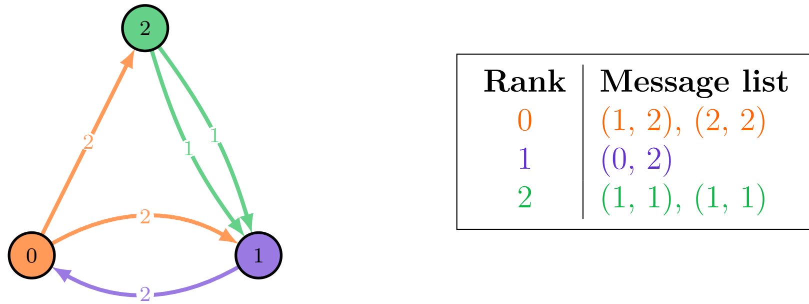
[1] This idea is very similar to the options `-fprofile-generate,-fprofile-use` of GCC, which gathers information about the execution patterns of a compiled program, then use this information to improve the quality of subsequent compilations.

Note that there can be duplicates t_i in the list if a rank sends multiple messages to the same target.

These lists of messages can be interpreted as the distributed adjacency lists of the underlying directed communication graph. With this information, we can build a weighted directed graph of the form :

$$G = (V, E) \quad \text{with} \begin{cases} V = \{0, 1, ..., N-1\} \\ E = \bigcup_{r=1}^{N} \{(r, t_i, w_i)\}_{i=1}^{m_r} \end{cases}$$

Where the edges $e \in E$ are triplets of the form (source, target, weight).



Rank	Message list
0	(1, 2), (2, 2)
1	(0, 2)
2	(1, 1), (1, 1)

Figure 3.1: *Example of a distributed adjacency list. Each rank r knows only its outward neighborhood, represented here by the different colors. The global communication graph is the union of all these local adjacency lists.*

The first step of the algorithm is to project the directed weighted communication graph into a simple undirected weighted graph. A *simple graph* is a graph without self-loops nor multiple edges. Intuitively, we want a simple graph because self-loops – representing self-communications – can simply be ignored and multiple edges can be summarized into a single edge representing the total communication between two processes. The requirement of undirected edges is both for performance reasons detailed in Appendix A and for compatibility with the MPI standard 4.1 [Message Passing Interface Forum, 2023], as explained by the following excerpt (p395, lines 43-45):

> *Advice to users.* Performance implications of using multiple edges or a nonsymmetric adjacency matrix are not defined. The definition of a node-neighbor edge does not imply a direction of the communication. (End of advice to users.)

The MPI standard assumes the communications to be symmetric and undirected *by default*. These assumptions allow a more scalable approach as it is then assumed that every rank knows its entire neighborhood. Our approach relaxes the symmetric assumption by developing a projection scheme allowing to "symmetrize" the communications by means

of edge weighting. This method allows increasing the quantity of information carried by the weight of each edge. At the end of our algorithm, our graph representation will respect the requirements of the MPI standard.

We thus develop a weighting scheme projecting the directed weighted graph G into its closest undirected weighted graph G'. As our end goal is the minimization of the communication time, we chose to design the weighting scheme such that it estimates the total communication time between every pair of rank. This estimation will provide the weight of the resulting undirected edge between each pair of vertices. Our methodology is the following:

1. Evaluate the time cost of each edge given its weight.
2. Combine the directed edges with the same source and target into a single directed edge. (i.e. multiple communications from one rank to another)
3. Combine the resulting directed edges into a single undirected edge between each pair of vertices.

3.1 Evaluation of the Time Cost of Single Messages

Let $G = (V, E)$ be the directed weighted communication graph. We define the weight of an edge w_{ij} as the size of the message sent by i to j, in bytes. To evaluate the time cost of a single message, we extend the standard latency/bandwidth model detailed for example in [Eijkhout et al., 2016]. To give an intuition, the latency is the delay between the data request and the data actually arriving. Bandwidth on the other hand is the rate at which data is transferred. As a crude analogy, latency is the time it takes you to bring your beverage to your mouth, whereas bandwidth is the rate at which you drink it.

This step can be done independently by every rank, as the time cost of a message only depends on the size of this message.

3.1.1 Standard Latency/Bandwidth Model

The standard model defined in [Eijkhout et al., 2016] assumes constant latency and bandwidth on a given machine. Using these two parameters, we can get a rough estimate of the time cost $T(w)$ of a message based on the machine's properties. The model evaluates the time cost of the edge (i, j) of weight w_{ij} using the equation :

$$T(w_{ij}) = L + \frac{w_{ij}}{B}$$

Unfortunately the assumption of constant latency and bandwidth does not hold in practice due to dynamic choices of communication protocol in the transport layer and network routing. This is clearly visible in Figure 3.2, which we will discuss shortly.

3.1.2 Extended Latency/Bandwidth Model

To mitigate the shortcomings of the standard model, we propose to generalize it to account for the message's size in the following natural way :

$$T(w_{ij}) = L(w_{ij}) + \frac{w_{ij}}{B(w_{ij})}$$

Where $L(w)$ and $B(w)$ are step-functions of the latency and bandwidth directly estimated on the supercomputer using the OSU benchmarks [MVA, 2024]. Values of this benchmark suite for Lucia and MeluXina are displayed in Figure 3.2.

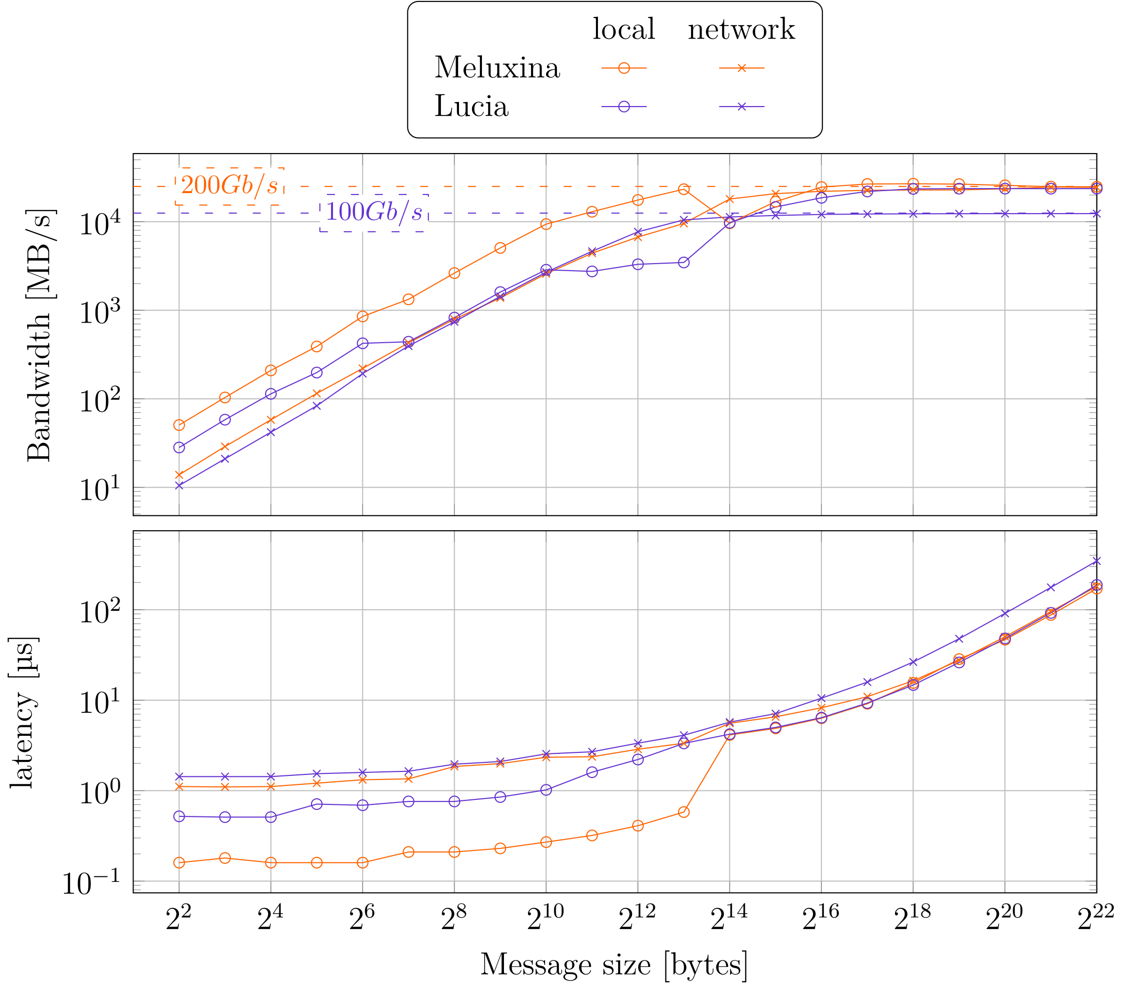


Figure 3.2: *Evolution of the bandwidth and latency with the size of the messages (in bytes). Values for local and network communication are shown for both the Lucia and MeluXina systems. Exact values can be found in Table B.1 and Table B.2*

As stated earlier, we clearly see on Figure 3.2 that latency and bandwidth are not constant in real-world systems. We observe that latency and bandwidth have opposite behaviors. Bandwidth grows linearly until a threshold located around 2^{14} bytes (16 kB), and then remain saturated. On the other hand, latency remains approximately constant until this threshold and then grows with the size of the message. Notice here that for Lucia, the maximal local bandwidth (—o—) is twice the maximal network bandwidth (—×—) whereas MeluXina has approximately the same maximal bandwidth for local and network transfers. On the other hand, latency remains approximately constant until this threshold and then grows with the size of the message. The behavior of the latency is actually due to the way OSU benchmarks are carried out, but it is still representative of real applications. Their official website states :

> ***osu_latency*** *- Latency Test*
> *The latency tests are carried out in a ping-pong fashion. The sender sends a message with a certain data size to the receiver and waits for a reply from the receiver. The receiver receives the message from the sender and sends back a reply with the same data size. Many iterations of this ping-pong test are carried out and average one-way latency numbers are obtained. Blocking version of MPI functions (MPI_Send and MPI_Recv) are used in the tests.*

Because they are using a combination of blocking calls and ping-pong repetition to measure latency, the receiver has to first receive the entire message before sending back the reply. Reaching the maximal throughput of the network means that we introduce an additional delay in the communication, coming from a combination of the injection time (the time it takes to send the message over the network) and reception time (the time it takes to receive the message in its entirety). This is why the latency grows after this threshold.

A more rigorous but equivalent way of taking this behavior into account is to expand the model as follows :

$$T(w_{ij}) = L(w_{ij}) + \frac{w_{ij}}{B(w_{ij})} + I(w_{ij})$$

With $I(w)$ the injection time of the message, which would be very small when the bandwidth is not saturated and grow linearly with the size of the message when it is. For notational simplicity, we simply group L and I into the same function L.

3.2 Aggregating Parallel Edges Having the Same Direction

The next step is to combine the directed edges with the same source and target into a single directed edge. This can still be performed independently on every rank. Indeed, this step consists in aggregating multiple edges in the local graph representation of each

rank. As they represent multiple messages from the same rank to the same target, we assume these messages to be sent sequentially. The time cost of the resulting edge is then the sum of the time cost of each individual message:

$$\tilde{w}_{ij} = \sum_{(i,j,w_{ij}) \in E} L(w_{ij}) + \frac{w}{B(w_{ij})} \quad \forall\, i, j = 0, 1, ..., N-1$$

The resulting distributed graph $\tilde{G} = (V, \tilde{E})$ is now a directed weighted graph for which each pair of rank has either no edge, a single directed edge or two directed edges with opposite directions. Each edge of this graph represent the total cost of the messages sent from one rank to another.

3.3 Aggregating Parallel Edges Having Opposite Directions

There remains only to combine the directed edges with opposite directions into a single undirected edge. Contrary to the previous steps, this cannot be done locally as it requires both the knowledge of the messages sent and received by the rank.

Two choices for synchronization are possible. Either we gather information about the local graph of each rank and perform the aggregation in a distributed manner, or we simply gather the entire graph on a single rank and aggregate the edges sequentially. We chose the latter because the partitioning steps discussed in Chapter 4 are sequential.

One might wonder why we do not simply use a distributed partitioning algorithm. The answer is simply because our use-case has exactly one vertex per rank. The amount of communication required to partition the graph would thus be much larger than the computation itself. The argument could be pushed further by asking why do we not perform the partitionning on 2 or 4 ranks for example. Our response to that argument comes from a more pragmatic point of view. As our algorithm is not supposed to be the time-consuming part of the user application, we prefer to keep it simple and easily extensible. Therefore, we gather the entire graph's topology on a single rank and aggregate multiple edges sequentially on this rank.

There are multiple choices for the aggregation of the edges depending on the application and the properties of the network. We describe two sensible choices here. At the end of this step, we obtain the final undirected weighted graph $G' = (V, E')$. Notice that the final set of edges E' is necessarily smaller than the original edge set E. Indeed, as we aggregate multiple edges into one and simply ignore self loops, we have $|E'| \leq \frac{N(N-1)}{2}$ whereas $|E|$ is unbounded. This has a direct computational impact on some of the steps of the partitioning algorithm.

3.3.1 The Case of a Half-Duplex Network

A half-duplex network is a network that can send data both ways but can only send or receive data at a time. In this case, we choose to sum the time cost of the two directed edges to accurately represent the sequentiality of the communications between the two ranks.

$$w'_{ij} = w'_{ji} = \tilde{w}_{ij} + \tilde{w}_{ji} \quad \forall\, i, j = 0, 1, ..., N-1$$

3.3.2 The Case of a Full-Duplex Network

A full-duplex network is a network that can send and receive data at the same time. In this case, we choose to keep only the maximum of the two directed edges as the time cost between the two ranks is driven by the slowest communication.

$$w'_{ij} = w'_{ji} = \max\left(\tilde{w}_{ij}, \tilde{w}_{ji}\right) \quad \forall\, i, j = 0, 1, ..., N-1$$

Chapter 4

Partitioning Methods

We remind that our objective is to find a perfectly balanced partitioning of the communication graph arising from the distributed computation problem. This means that given a communication graph $G = (V, E)$ with edge weights $w(u, v) = w_{uv}\ \forall(u, v) \in E$ as produced by the methodology described in Chapter 3, we look for a k-way partitioning P respecting the following perfect balance constraint:

$$|P_i| = |P_j| = \frac{|V|}{k} \qquad \forall i, j \in \{1, 2, ..., k\}$$

This constraint arises from the physical problem. The nodes of a supercomputer have the same number of cores[1] and can thus host the same maximum number of ranks. While theoretically possible to request more resources than needed for the application (i.e. letting some cores idle), it is financially undesirable. Supercomputers billing is based on the numbers of allocated cores or nodes. Using fewer resources than requested equates financial losses. Hence, we aim to find a perfectly balanced partitioning.

As stated in the introduction, our goal is to minimize communication costs. Figure 3.2 displayed both the local and network latency/bandwidth for two systems, we observed in these charts that the time cost of a message is greater when sent over the network. We thus have an incentive to minimize network communications. These messages are represented by edges with endpoints in different partitions, one choice of objective value is the sum of the weight of these edges.

$$cut(P) = \frac{1}{2} \sum_{v \in V} \sum_{u : P(u) \neq P(v)} w_{uv}$$

This objective value represents the total cost of network communications, as if all communications happened sequentially. For more than two cluster-nodes, this objective

[1] This is technically not true, as supercomputer can have nodes with different number of cores. But to the best of our knowledge it is not possible to mix two different type of nodes in a single resource request.

overestimates the true cost of communication. Indeed, two different pairs of nodes can exchange communications simultaneously, meaning that our metric would double the true cost of communication. However, this metric provides a global approximation of the network volume while being easy to compute and update.

Our final optimization problem can be stated as follows :

$$\min_{P} \quad cut(P) \tag{4.1}$$

$$\text{s.t:} \quad |P_i| = |P_j| = \frac{|V|}{k} \qquad \forall i, j \in \{1, 2, ..., k\} \tag{4.2}$$

$$\bigcup_{i=1}^{k} P_i = V \tag{4.3}$$

$$\bigcap_{i=1}^{k} P_i = \emptyset \tag{4.4}$$

Constraint (4.2) is the perfect balance constraint whereas (4.3) and (4.4) ensure that P respects the definition of a partition. Note that this problem is NP-Hard and can thus only be solved approximately through heuristics [Garey et al., 1976].

To solve this problem, [Hendrickson and Leland, 1995b] proposed a general multi-level framework that has been widely adopted by the partitioning community. This framework allows to increase extensibility and modularity[2] of the chosen methods as well as improving the overall efficiency on very large graphs. In this chapter, we detail this framework as well as possible heuristics for each of the three phases.

We insist on the fact that we do not have any positional or geometric information attached to the graph that would allow us to leverage spatial techniques such as geometric partitioning [Gilbert et al., 1998]. We only rely on topological information.

4.1 The Multi-Level Framework

Let $G^0 = (V^0, E^0)$ be an undirected graph with weighted edges and weighted vertices. The multi-level graph partitioning framework consists of three sequential steps. First, the graph is *coarsened* by iteratively contracting edges and merging vertices to construct the graph of the next level. Mathematically, we construct the graph at level $l+1$, $G^{l+1} = (V^{l+1}, E^{l+1})$ from the graph at level l : $G^l = (V^l, E^l)$ such that the vertices of the coarsened graph $V^{l+1} = \{V_1^{l+1}, V_2^{l+1}, ..., V_{N_{l+1}}^{l+1}\}$ form a partition of the vertices V^i of the current graph :

$$\bigcup_{j=1}^{|V^{l+1}|} V_j^{l+1} = V^l \qquad \bigcap_{j=1}^{|V^{l+1}|} V_j^{l+1} = \emptyset$$

[2] Modularity here is not intended as a graph-theoritic metric but in the literal meaning.

This means the vertices of the graph at level $l+1$ represent multiple vertices of the finer graph at level l. As we are working with weighted graphs, we define the weight of a vertex in a coarser graph to be the sum of the weights of the vertices it represents in the finer graph.

$$c^{l+1}(V_j^{l+1}) = \sum_{u \in V_j^{l+1}} c^k(u)$$

Similarly, the weight of the edge (V_j^{l+1}, V_k^{l+1}) in the coarsened graph at level $l+1$ is obtained by summing the weights of the edges (u, v) in the graph of level l such that $u \in V_j^{l+1}$ and $v \in V_k^{l+1}$.

$$w^{l+1}(V_j^{l+1}, V_k^{l+1}) = \sum_{u \in V_j^{l+1}} \sum_{v \in V_k^{l+1}} w^l(u, v)$$

Edges resulting in a self-loop can either be removed or kept, depending on the coarsening algorithm used.

The coarsening step stops once the number of vertices of the graph reaches an arbitrary threshold, usually chosen as a function of the number of partitions k, e.g. $|V^l| \leq 64k$.

Although our final partitioner does not perform the coarsening phase, we explain briefly some possible algorithms for this step and why they fail in our case. Note that the source code of the partitioner contains an empty coarsening step in which we can easily implement coarsening strategies, or call external libraries performing this task.

The second step is the *initial partitioning*, in which we perform an initial k-way partition of the smallest graph, as described in the introduction of this chapter. Note that this initial partitioning is allowed to break the balancing constraint (4.2).

The last step is the *uncoarsening*, during which we expand the edges contracted during the coarsening phase and propagate the initial partition to the finer graphs of the hierarchy by assigning the partition of the vertex in the coarser graph at level $l+1$ to all the vertices it represents in the finer graph at level l.

$$\forall V_j^{l+1} \in E^{l+1} \; : \; P(V_j^{l+1}) = p \implies P(V_k^l) = p \quad \forall V_k^l \in V_j^{l+1}$$

The strength of this step lies in its ability to perform local optimizations at each iteration, allowing to increase the quality of the partition as well as enforcing the balance constraint that may have been broken during the initial partitioning.

Figure 4.1 displays a visual representation of the different steps of the multi-level framework.

Figure 4.1: *Example of a complete partitioning using the multi-level framework. Note that for clarity, the weights of the vertices is displayed through their size.*

4.1.1 Coarsening

We now describe a few possible possibilities of coarsening strategies. First, it is important to understand what defines a good coarsener. A good coarsener needs to :

- Efficiently reduce vertices at each iteration. This efficiency is twofold, minimizing computational cost and maximizing vertex reduction. A coarsening algorithm that contracts only a few edges per iteration is considered *stalled*. Stalling leads to a longer hierarchy, hindering the subsequent uncoarsening step.

- Contract high costs weights.

The second point is a benefit towards the objective function. Indeed, one can easily see that the sum of weight of all edges can be expressed as the sum of the edges internal to

the partitions and the edges crossing partitions :

$$\sum_{(u,v)\in E} w_{uv} = \sum_{P(u)=P(v)} w_{uv} + \sum_{P(u)\neq P(v)} w_{uv} = I(P) + cut(P)$$

As the total sum is constant, maximizing the internal cost $I(P)$ and minimizing the edgecut is the same problem. Therefore, a coarsener contracting heavy edges will benefit the objective. Two classes of heuristics achieving this goal are possible: matching-based coarsening and clustering-based coarsening.

Matching-Based Coarsening

This class of coarsener uses matchings to choose the edges it will contract at each iteration. A *matching* is a subset of edges $\tilde{E} \subseteq E$ that do not share vertices. In other words, each vertex of the graph is incident to at most one edge of the matching. Remark that the set of edges $\tilde{E}$ is by definition a set of independent edges, the second objective of a coarsening algorithm can thus be achieved by finding *maximal independent sets of edges*.

A very simple yet powerful heuristic is the Randomized Heavy-Edge Matching (RHEM) first introduced by [Karypis and Kumar, 1998] and described in Algorithm 1. In words, the algorithm first marks every vertex as "unmatched". Then it visits the remaining unmatched vertices in random order (chosen prior to the search) and finds the edge with the highest weight linking the visited node with one of its unmatched neighbors. If such an edge exists, it is added to the matching and both vertices are marked as "matched".

Algorithm 1 Randomized Heavy-Edge Matching

```
Require: G = (V, E)
  M ← ∅, Ẽ ← ∅
  for v ∈ shuffle(V) do
     if v ∉ M then
        u ← argmax_{u∉M} w_vu          ▷ Find the unmatched edge of maximum weight
        if u ≠ ∅ then
           M ← M ∪ {u, v}              ▷ Those vertices are matched
           Ẽ ← Ẽ ∪ {(u, v)}            ▷ Add the edge to the matching
        end if
     end if
  end for
```

The RHEM algorithm has the benefit of being efficient in terms of complexity, with a worse case runtime of $\mathcal{O}(|E|)$ and with a clear goal of maximizing the internal weights.

However, it is important to note that the RHEM heuristic has a tendency to stall if the degree distribution of the vertices is highly skewed. This is the case for social networks,

displaying a power law degree distribution [Mislove et al., 2007]. In such networks, some vertices have a very high degree whereas the majority of the vertices have a rather low degree. This makes the cardinality of independent edge sets small, leading to only a fraction of the vertices getting merged. This is not a problem displayed by our use-case, the graphs issued from scientific applications (e.g. Finite differences, Finite Element Methods, numerical linear algebra) are highly regular and therefore lead to a very narrow degree distribution. A possible heuristic for power law networks is discussed in section 4.1.1. The interested reader can also consult [Lasalle et al., 2015, Meyerhenke et al., 2014].

Clustering-Based Coarsening

This class of coarsener can be seen as a generalization of the matching-based coarsening as **they are able to merge more than two vertices at once** by leveraging a *size-constrained clustering problem.*

As described in section 2.1, a graph clustering problem attempts to maximize the modularity Q of its clustering. The optimization problem reads

$$\begin{aligned} \max_P \quad & Q(P) = \frac{1}{2m} \sum_{u \in V} \sum_{v \in V} A_{uv} \frac{k_u k_v}{2m} \delta(P(u) = P(v)) \\ & \bigcup_{i=1}^{k} P_i = V \\ & \bigcap_{i=1}^{k} P_i = \emptyset \end{aligned}$$

With A the adjacency matrix of the graph, $m = \sum_{(u,v) \in E} w_{uv}$ the sum of the weight of all edges, $k_u = \sum_{(u,v) \in E} w_{uv}$ the weighted degree of the vertex u and $\delta(P(u) = P(v))$ an indicator function whose value is 1 if u and v belong in the same partition and 0 otherwise.

Several efficient algorithms have been designed to solve this problem. The oldest is Newman's Fast Algorithm [Newman, 2004]. The Louvain algorithm [Blondel et al., 2008] builds upon Newman's algorithm by implementing a multilevel component. Finally, the Leiden algorithm [Traag et al., 2019] improves the Louvain algorithm by adding guarantees on the connectivity of the resulting partitions, at the cost of a significantly more complex implementation (per their words).

However, these algorithms display the same problem when applied for a partitioning setting. **As they do not enforce balance constraint, the resulting coarsening is subject to high imbalance in the vertex weight**. Although we have allowed the initial partitioning step to violate the balance constraint (4.2), diverging too much from it puts unnecessary pressure on the uncoarsening step, which would have to restore perfect balance.

To mitigate this problem, [Meyerhenke et al., 2014] proposed to add a size constraint to the clustering algorithm. The authors apply their method to the label propagation algorithm and called the result "size-constrained label propagation". We propose to test this method on the Louvain algorithm.

The classical Louvain method is described by the pseudocode in Algorithm 2.

Algorithm 2 Classical Louvain Algorithm

Require: $G = (V, E)$
1: $P(u) \leftarrow u \forall u \in V$ ▷ Initial clustering
2: **while** True **do**
3: $P \leftarrow \text{LouvainStep}(G)$
4: **if** P has not changed **then**
5: **return** P
6: **else**
7: $G \leftarrow \text{contract}(G, P)$ ▷ Contract each partition into a single vertex
8: **end if**
9: **end while**
10: **function** LouvainStep(G)
11: $P(u) \leftarrow u \forall u \in V$
12: **while** True **do**
13: $P' \leftarrow P$
14: **for** $u \in V$ **do**
15: $P(u) \leftarrow \text{argmax}_p \Delta Q(u, p)$ ▷ Move u to the partition that maximizes the gain in modularity
16: **end for**
17: **if** $P = P'$ **then**
18: **return** P
19: **end if**
20: **end while**
21: **end function**

Intuitively, the Louvain algorithm operates by level. At each level, it visits the vertices in random order and moves them to the partition maximizing the gain in modularity until no vertex can be moved to a partition increasing the modularity. The algorithm then computes the quotient graph of the partition and restarts the process. A *quotient graph* is a graph whose vertices are the partitions of the original graph and whose edges are the edges between the partitions. The algorithm stops when we obtain the same partitioning twice in a row.

The size-constrained Louvain algorithm adds a constraint on the size of the partitions by upper bounding the number of iterations at each level. Practically, this simply translates

to adding a maximum number of iterations of the loop starting in line 12 of Algorithm 2. We also added a simple algorithmic improvement to the classical Louvain algorithm. We noticed that the original algorithm visits every vertex of the graph at each iteration of the inner loop (line 14-16 of the above pseudocode). This is not necessary, as the gain of a vertex only depends on its neighbors, vertices for which no neighbors have been moved are guaranteed to remain in their partition. Therefore, instead of visiting every vertex at each iteration, we use a queue that initially contains all the vertices of the graph. Every time a vertex is moved, we add its neighbors to the queue if they are not already in it. Using this simple trick, we are certain to visit only the vertex that are likely to be moved.

4.1.2 Initial Partitioning

We now delve into the initial partitioning. Naturally, a good partitioner should produce a partitioning with the lowest edgecut. Multiple algorithms have been designed for this objective. We name a few of them and motivate our choice for the final algorithm.

Recursive Bisectioning (RB)

The first class of algorithms is the *recursive bisectioning* algorithms. These algorithms work by recursively dividing the graph into two approximately equal parts until the desired number of partitions is reached. If the number of desired partitions is not a power of two, the algorithm can be adapted by modifying the target size of the two resulting partitions (e.g. 30% and 70% of the original graph instead of 50/50). This algorithm is conceptually simple and generally produces acceptable results. However, it has one drawback that made us look for other algorithms.

Look-Ahead Bias: RB is a greedy algorithm. It attempts to find the best 2-way partition at each step, but doing this may lead to poor results in the subsequent steps. This behavior arises even under the assumption of a perfect bisectioner, as it is intrinsic to the greedy recursive nature of the algorithm.

The bisection at each step can be performed with, for example, Spectral Bisection. The spectral bisectioning leverage spectral properties of the Laplacian of the graph $L = A - D$ with A the adjacency matrix and D the (diagonal) weighted degree matrix. It uses the eigenvector corresponding to the second-smallest eigenvalue, called the Fiedler vector [Fiedler, 1975][3]. It then assigns partitions to the vertices based on the sign of the corresponding component of the Fiedler vector.

[3]The smallest eigenvalue being 0 because the Laplacian is semi-positive definite by definition.

Direct k-way Partitioning

We turn to another class of algorithms, namely the *direct k-way partitioning*. As the name suggests, these algorithms directly partition the graph into k partitions. We have chosen this class of algorithm to avoid the look-ahead bias inherent to the recursive bisectioning.

We implement a *k-way greedy* algorithm, which starts by placing every vertex in an "imaginary" partition (with ID of -1) and iteratively finds the vertex-partition pair with the smallest increase in objective function and moves the vertex to this partition. During our research, we found that our idea was similar to the algorithm proposed by [Predari and Esnard, 2016], called KGGGP, which stands for K-way Greedy Graph Growing Partitioning. The authors use KGGGP with the goal to partition graph with fixed vertices and add a connectivity constraint to their algorithm. As we suppose our graphs to be (near)-regular, we do not take this constraint into account.

Before delving into the details of the algorithm, we define the concept of *gain*. The gain $g_v(P_j)$ of the movement of a vertex v from its current partition to a partition P_j is defined as the difference between the edges linking v to its current partition and the edges linking v to the partition it is moved to. Mathematically, this translates to

$$g_v(P_j) = \underbrace{\sum_{u \in P_j} w_{vu}}_{\text{Edges towards } P_j} - \underbrace{\sum_{u:P(u)=P(v)} w_{vu}}_{\text{Edges internal to } P(v)}$$

Notice that the gain function is consistent, we can readily check that $g_v(P(v)) = 0$. This detail might sound trivial, but notice that if we had a gain function such that $g_v(P(v)) > 0$ for some v, then any algorithm would be able to improve the objective function indefinitely.

The gain values can be computed up front for the whole graph and updated after each movement by generalizing the data structure introduced by [Fiduccia and Mattheyses, 1982]. Indeed, Fidduccia and Mattheyses introduced a *gain bucket* structure to store the gains of the movement of each vertex. In their work, the authors only considered the 2-way partitioning problem (bisectioning), the gain of each vertex was thus simply the gain of moving to the other partition. Their original data structure is represented in solid lines on Figure 4.2. One can see that this structure allows for quick retrieval of the movement with the highest gain by simply taking the first element of the bucket at index $MaxGainBucket$. Furthermore, it allows an update of the gain g_v in constant time by using the pointer at index v of the lookup table. We remind that as they only considered bisectioning, the gain g_v is defined as moving v to the only other partition. A detailed overview of the structure is given in [Fiduccia and Mattheyses, 1982].

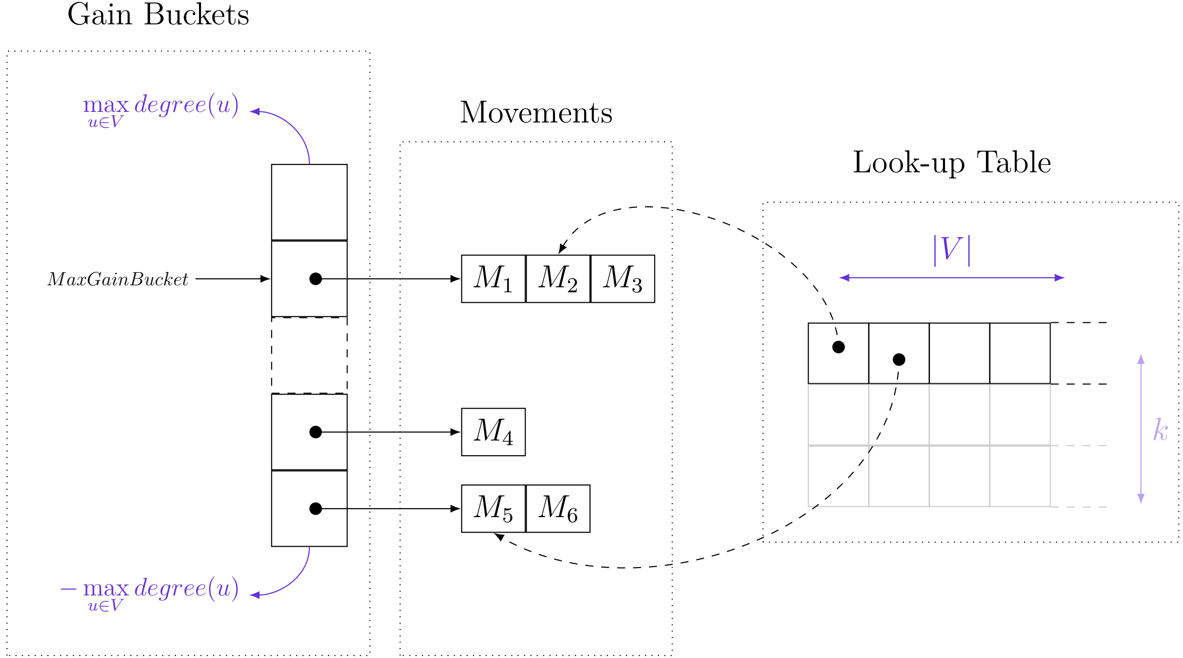


Figure 4.2: *Representation of the gain bucket structure as defined by Fidduccia and Mattheyses.*

We extend this structure to the k-way partitioning problem by simply adding a dimension to the lookup table of the movement, shown in gray lines in Figure 4.2. In this way, we can quickly find the movement with the highest gain across all possible movements of vertices and partitions.

Our algorithm, shown in Algorithm 3, then simply initializes the gain bucket with the cost of moving each vertex to each partition. As we initially place every vertex into the imaginary partition, the initial bucket can be initialized with $g_v(P_j) = k_v$ where we remind that k_v is the weighted degree of the vertex v. The algorithm then iteratively retrieves the movement with the highest gain and applies it only if the vertex has not already been assigned to a partition. The algorithm stops when all vertices have been assigned.

One can see that the initialization has a cost of $\mathcal{O}(k|V|)$, which can become expensive whenever k is high. A cheaper initialization method is to randomly choose k initial vertex, assign them each to a partition and update only the gain of their neighbors in the data-structure. The cost of this initialization method drops to $\mathcal{O}(k \max_{v \in V} \deg(v))$, which is a notable improvement. Unfortunately the improvement in complexity is associated with a loss of information. Indeed, by doing this, we loose global information on the graph. Notice that our first initialization method will always start by assigning the vertices with the lowest weighted degree to the partitions. In a scientific communication graph, these vertices generally correspond to the boundaries of the domain. The second, faster

method, will not factor this information into account and might initially place the different partitions close to each other.

Algorithm 3 k-way greedy partitioning

```
1:  P(v) ← −1   ∀v ∈ V                         ▷ Initialize all vertices to an imaginary partition
2:  g_v(P_j) ← Σ_{u∈V} w_{vu}   ∀v ∈ V, j ∈ {1, 2, ..., k}     ▷ Initialize the gain bucket
3:  while P_{−1} ≠ ∅ do
4:      v, j ← argmax_{v,j} g_v(P_j)            ▷ Find the movement with the highest gain
5:      if P(v) = −1 then
6:          P(v) ← j
7:          for u ∈ N(v) do                     ▷ Update the gain of the neighbors
8:              g_u(P(v)) ← g_u(P(v)) − w_{uv}
9:              g_u(P_j) ← g_u(P_j) + w_{uv}
10:         end for
11:     end if
12: end while
```

We can observe that updating the neighbors' gain is as simple as adding and subtracting the weight of the edge linking them to the moved vertex. This update is made efficient by the usage of the heap structure described in Figure 4.2. Figure 4.3 displays a visual representation of the gain update of a neighbor.

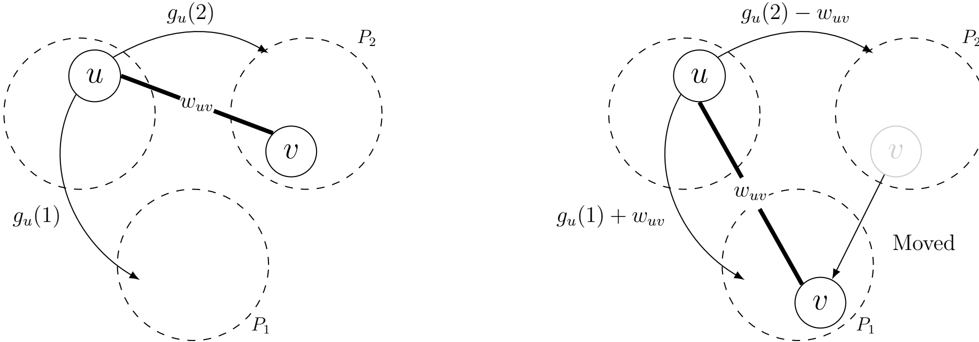


Figure 4.3: *Visual representation of the gain update. In this chart, v is moved from partition 1 to 2. The gain of its neighbor u is thus updated to reflect the movement.*

4.1.3 Uncoarsening

We now enter the final step of the multi-level framework, the uncoarsening. The goal of this step is twofold. First, it aims at recovering the balance of the partitioning which may have been broken during the initial partitioning discussed above. Second, it performs local optimizations by moving vertices from one partition to a neighboring partition in

order to improve the objective value. The way it applies these two goals is by working in the opposite direction of the coarsening step. During coarsening, we contracted edges and merged vertices, here we expand the edges contracted during coarsening.

First, the partitioning P^l of G^l is obtained by assigning the partition of the vertex in G^{l+1} to all the vertices it represents in G^l. As every vertex of the level $l+1$ represents a set of vertices of the level l, we write :

$$\forall u \in v : P^l(u) = P^{l+1}(v) \quad v \in V^{l+1}$$

The pseudocode is given in Algorithm 4. After the projection, we apply the refinement algorithm and iterate onto the next level of the hierarchy.

Algorithm 4 Projection of the partitioning

Require: $G^{l+1} = (V^{l+1}, E^{l+1}), G^l = (V^l, E^l), P^{l+1}$
1: **for** $v \in V^{l+1}$ **do**
2: $\quad P^l(u) \leftarrow P^{l+1}(v) \quad \forall u \in v$
3: **end for**
4: **return** P^l

The refinement algorithm commonly used in practice by partitioners is the FM algorithm [Fiduccia and Mattheyses, 1982]. **Although this algorithm and its variant are very efficient, they do not provide any guarantee on the balance constraint. This is a problem for our use-case.** Put intuitively, our final objective is to put the ranks of each partition on a cluster-node of the supercomputer. If the partitions are not perfectly balanced we will have more ranks then we can physically fit on a cluster-node, resulting in an undefined behavior: the reordering algorithm will not fail, but the result may be very far from the output of the partitioner.

Negative Cycle Detection

To provide guarantees on the perfect balance constraint, we use a variant of the negative-cycle detection algorithm proposed by [Sanders and Schulz, 2012]. This algorithm leverages information about the quotient graph to allow both non-trivial vertex movements and balance improvement. Contrary to other commonly used refinement algorithms such as KL/FM which can only perform one movement at a time, this algorithm can move multiple vertices at once between an arbitrary number of partitions. To the best of our knowledge, this is the only algorithm capable of such complex movements. As we delve deeper into the details of this algorithm in the rest of this section, let us first provide a general overview of its steps.

1. As long as the balance constraint is not satisfied, or we improve the objective, we iterate over steps 2-4.

2. Perform a local search on the vertices to find high-gain movements between partitions. If multiple movements are possible from partition i to partition j, keep only the best one.

3. Using the movement found in (2), construct a weighted directed quotient graph. The vertices of the quotient graph are the partition and the edges are the movements from one partition to another. These edges are directed, and we assign a weight equal to the opposite of the gain of the movement they represent.

4. Find a negative cycle in this quotient graph. If such a cycle exists, we can perform the corresponding movements. Notice that the number of vertices in the cycle gives the number of partitions involved in the movement. If no such cycle exists, we can still use the computed information to to find a shortest path in the graph and further improve balancing.

Some of these steps might sound abstract at first. We hope the rest of this section will make them clearer.

Local Search

The goal of the local search is to provide good candidate vertices to move from one partition to another. To avoid the burden of updating the gain of all the neighbors of moved vertices, only the vertices that are not adjacent to a moved vertex are eligible during the local search. To help enforce this, we keep track of the moved vertices.

To find the candidate movements, we iterate over all pairs of partitions $(P_i, P_j), i \neq j$ in a random order and keep only the movement of an eligible vertex with the highest gain. Mathematically, we compute the following set :

$$S = \{(P_i, P_j, \text{argmax}_{v \in P_i} g_v(P_j)| \quad i, j \in \{1, 2, \ldots, k\}, i \neq j\}$$

To prevent any neighbor of a chosen vertex v to be chosen during the rest of the local search, all neighbors to v are removed from the set of eligible vertices.

Algorithm 5 Local Search

Require: the current partitioning P, the gains g_v, the moved vertices M

1: $S \leftarrow \emptyset$
2: $E \leftarrow V \setminus (M \cup (\bigcup_{u \in M} N(u)))$ ▷ Set of eligible vertices
3: **for** $(i, j) \in \{1, 2, \ldots, k\}, i \neq j$ **do**
4: $\quad v \leftarrow \text{argmax}_{v \in E \cup P_i} g_v(P_j)$
5: $\quad S \leftarrow S \cup \{(P_i, P_j, g_v(P_j))\}$ ▷ Store the movement
6: $\quad E \leftarrow E \setminus (\{v\} \cup N(v))$ ▷ Remove the neighbors of v from the eligible set
7: **end for**
8: **return** S

Construction of the Quotient Graph

Each movement $(P_i, P_j, g_v(P_j)) \in S$ obtained by the local search can be seen as a directed edge from partition P_i to P_j. We can then construct the weighted directed quotient graph :

$$Q = (V_Q, E_Q) \quad : \quad \begin{cases} V_Q = \{P_1, P_2, \ldots, P_k\} \\ E_Q = \{(P_i, P_j, -g_v(P_j)) \mid (P_i, P_j, g_v(P_j)) \in S\} \end{cases}$$

Notice that we set the weight of the edges to the opposite of the gain of the corresponding movement. Intuitively, this allows the negative cycle detection algorithm to "get stuck" on negative cycles, corresponding to a combination of movements improving the objective value.

There remain a small detail to address. The benefit of this will be clearer in the following section. In order for the negative cycle detection algorithm to work efficiently, we add two imaginary vertices to the quotient graph : s, the source and t, the target. We then add weightless edge (thus with 0 weight) from s to every **overloaded partition** and from every **underloaded partition** to t. An overloaded partition is a partition with more vertices than allowed by the balance constraint whereas an underloaded partition has fewer vertices than required.

$$O = \left\{(s, P_i, 0) \mid i : |P_i| > \frac{|V|}{k}\right\} \qquad U = \left\{(P_i, t, 0) \mid i : |P_i| < \frac{|V|}{k}\right\}$$

The final quotient graph is then $Q = (V_Q \cup \{s, t\}, E_Q \cup O \cup U)$.

Negative Cycle Detection

The final step of the refinement algorithm is to apply negative cycle detection on the resulting quotient graph Q. We use the well-known Bellman-Ford algorithm, starting at the vertex s. Due to the weighting of the edges, a negative cycle in this graph corresponds

to a movement of the vertices improving the objective value. Furthermore, the total improvement is equal to the sum of the weights of the cycle.

After applying the negative cycle detection algorithm, we differentiate two cases: either we find a negative cycle or we don't.

If we don't find a negative cycle, we can use the distances computed from s to t by the Bellman-Ford algorithm to find the shortest path in the graph. In this case, one partition will lose exactly one vertex, that will be gained by another partition. This movement will improve the balance of the partitioning while minimizing the increase in objective value.

If we find a negative cycle, we apply all movements corresponding to the edges of the cycle. In this case, every partition taking part in the movement gives exactly one vertex to the next partition in the cycle and receives exactly one vertex from the previous. The size of each partition is thus left unchanged by the movement. Finding a negative cycle thus means we find a vertex movement improving the objective value without modifying the balance constraint. At the end of the movement, every vertex moved is marked as such. Meaning that neither these vertices nor their neighbors will be eligible during the next iteration of the local search. The astute reader will have noticed that in the case of an already perfectly balanced partitioning, the set of edges O linking the imaginary source s to the quotient graph is empty. Therefore, the negative cycle detection will never find a cycle. A solution to handle this special case is simply to start at another vertex of the graph. For this purpose, we choose to start at the vertex with the highest degree of the quotient graph. In this manner, we are guaranteed that the starting vertex is not an independent vertex.

We would like to note that this special ill-defined case is a result of our modification of the algorithm proposed by [Sanders and Schulz, 2012]. In their algorithm, the authors completely separate the negative cycle detection from the shortest path computation. The original algorithm first tries to find a negative cycle in the quotient graph **without adding the imaginary vertices s and t.** If they find a negative cycle, they proceed as we explained above. If they don't, they add the imaginary vertices and compute the shortest path **as they are now certain the graph does not contain a negative cycle.** Thus, the original algorithm does not suffer from the special case we described above, but this comes at twice the cost of the algorithm we presented.

As we are interested in efficiency, we chose to merge the two steps, allowing the reuse of the information computed during the negative cycle detection.

Chapter 5

From Partition to Final Ordering

In Chapter 3, we discussed how to create the communication graph of the user application. We then tried to minimize the network communications by partitioning the graph in Chapter 4.

In this chapter, we explain the second part of the reordering library. Its objective is to create a mapping from the initial rank allocation provided by the resource manager to the final allocation given by the partitioner.

In the general case, this task requires a collective synchronization. We show that under a non-restrictive assumption on the initial allocation, the problem can be solved in reverse with simple point to point messages.

As stated in section 2.2, we usually use the terms "processes" and "ranks" interchangeably because the rank is defined as the unique identifier of a process in a given MPI communicator. Therefore, given a communicator the rank allows to unambiguously identify a process. However, this is not true in this chapter as we will modify the identifiers (ranks) of the processes. We now use the terms "processes" and "ranks" with their literal meaning. Indeed, in our use-case a process is physically bound to its core and cannot be moved. Therefore, we can only modify the ranks of the processes to fit the final partitioning. It is critical to understand the following nuance :

We are reordering the ranks by modifying the identifiers of the processes, we are not relocating the processes themselves.

We emphasize this because it is actually the source of many bugs in user code. Indeed, a communication between two **ranks**, say 0 and 3 in the initial communicator will not result in the same data transfer after reordering. We highlight this behavior visually in Figure 5.1. **Knowing this, it is mandatory that the user loads their data after**

reordering has been performed. If their data is loaded before the reordering, the user is responsible for updating the data to reflect the reordering, otherwise their applications will result in an undefined behavior.

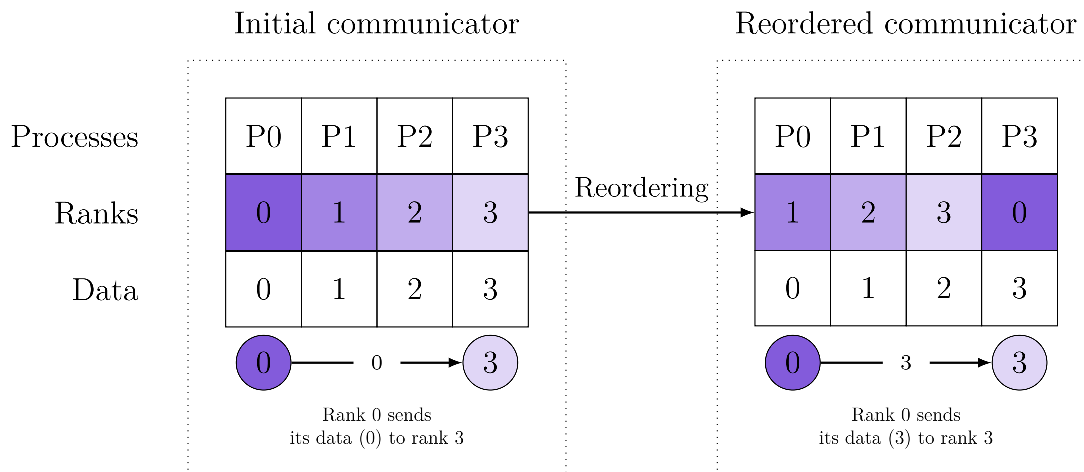


Figure 5.1: *Effect of reordering on data exchange. As processes are not moved, if the data is loaded before reordering, the communications will not result in the intended data exchange.*

5.1 Mapping the Partitions

Figure 5.2 displays an example of an arbitrary initial allocation with 6 ranks on 2 nodes as well as the final partitioning obtained by the partitioner. We will use this small example to highlight some key aspects of the problem.

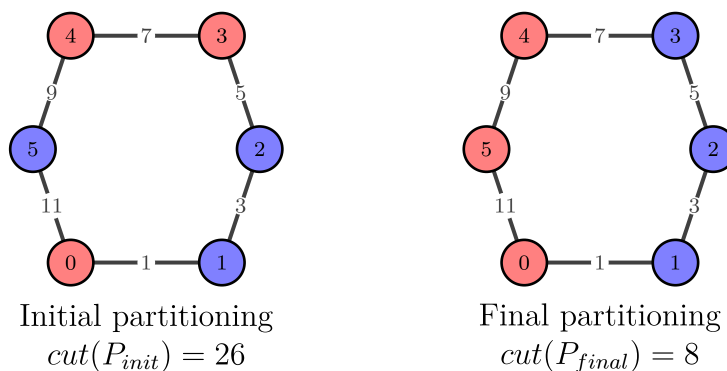


Figure 5.2: *Example of a communication graph with the initial allocation (left). The target allocation is given by the partitioning (right).*

At first, this problem seems trivial. One might say, looking at Figure 5.2 that we "*only need to move 5 from blue to red partition and 3 from red to blue*". This section will describe an algorithm that draws the same conclusions by computing the complete mapping. We remind the reader that each vertex of the graphs represents in reality a separate process that can only access the data of other processes by sending and receiving messages. The algorithms described here are thus distributed. Additionally, a process cannot be easily relocated to a different node. Therefore, each process in the initial allocation will have to change its own rank to fit the final allocation without conflict with any of the other ranks. **This kind of consensus can actually be seen as finding a partition-preserving isomorphism between the two allocations.**

> A *partition-preserving isomorphism* between two graphs $G = (V, E)$ with partitioning P and $G' = (V', E')$ with partitioning P' is a bijective function
>
> $$f : V \mapsto V' : v \mapsto f(v)$$
>
> Such that : $P(i) = P(j) \implies P'(f(i)) = P'(f(j)) \; \forall \, i, j \in \{0, 1, ..., |V| - 1\}$
>
> Note that this mapping leaves the edges sets E, E' unchanged.

In other words, it means the partitions are invariant with regard to the mapping f. This is exactly what we are trying to achieve. By finding the mapping f from the initial to the final partitioning, we are conserving the shared (physical) partitions of the supercomputer while providing an ordering of the ranks leading to a better edgecut. The partition-preserving isomorphism for our example problem is shown in Figure 5.3.

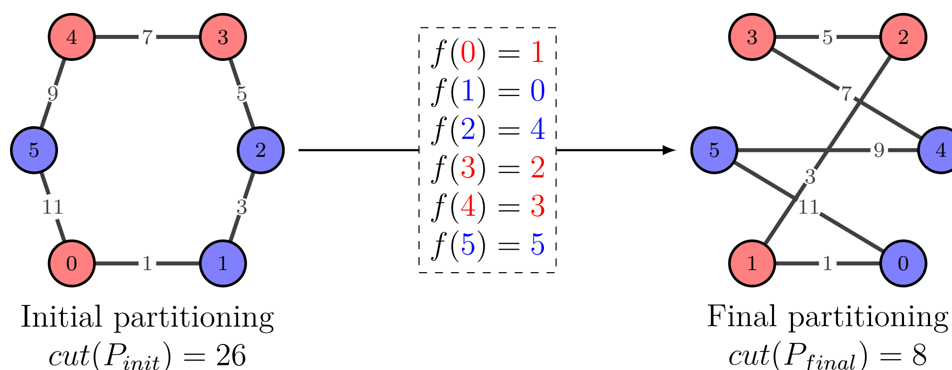


Figure 5.3: *Partition-preserving isomorphism $f(.)$ between initial and final partitioning. Contrary to Figure 5.2 we have graphically kept the partitions unmoved to fit the reality of the problem.*

5.1.1 Finding the Isomorphism

Finding the isomorphism can be seen as a reverse lookup problem[1] as we show here. Let us first introduce some useful values and notations. To each rank r, we assign the following values :

The (unique) ID of the partition r belongs to :	$P(r)$
The (unique) rank of r in its partition :	$R(r) = \|\{i : P(i) = P(r), i < r\}\|$

We add a superscript I or F to distinguish between initial and final partitioning. We thus have two pairs of values for each rank r :

$$\begin{cases} I(r) = (P^I(r), R^I(r)) \\ F(r) = (P^F(r), R^F(r)) \end{cases}$$

We also use this notation to design the initial and final graphs and partitions, thus denoted by G^I, G^F, P^I, P^F.

To be clear r is the rank of the process in the application, whereas the rank of r in the partition, $R(r)$, is the number of processes in this partition with a rank lower than r. We will soon give an example to clarify this[2]. The partition IDs are given arbitrarily, the only constraint is that they are unique. **Uniqueness matters because we are trying to construct a bijective mapping.**

The isomorphism $f(r) = r'$ can then be constructed by finding the pairs (r, r') such that $I(r) = F(r')$. This is the reverse lookup operation. Each rank r has to find the rank r' such that $I(r) = F(r')$, which corresponds to finding the index of an element in a set.

[1] A reverse lookup is the action of searching the index of an item in a set. For comparison, a lookup is the action of looking up the item at a given index in a set.

[2] For the more MPI inclined, the partition (initial or final) is viewed as a `MPI_COMM_TYPE_SHARED` communicator. Therefore, process with rank r in `MPI_COMM_WORLD` simply has another rank in the shared communicator.

Proof: if $I(r) = F(f(r))\ \forall r$***, then*** $f(r)$ ***is a partition-preserving isomorphism.***

Let i, j be any two ranks on the same initial partition : $P^I(i) = P^I(j)$. Then, if

$$I(r) = F(f(r)) \quad \forall r$$

holds, we have

$$P^I(i) = P^F(f(i))$$
$$P^I(j) = P^F(f(j))$$

Therefore, as we supposed $P^I(i) = P^I(j)$, we obtain $P^F(f(i)) = P^F(f(j))$.

As this is true between any two ranks on a given partition, f is partition-preserving.

Note that this condition is sufficient but not necessary. Indeed, the proof does not make use of the values of R^I and R^F. This is because any permutations of the ranks inside a given partition remains a valid partition-preserving isomorphism. We chose to fix $R^I(r) = R^F(f(r))$ to be able to design an algorithm providing a unique, deterministic solution.

The Table 5.1 shows the values discussed above for the example shown in Figure 5.3.

	$I(r)$		$F(r)$		
r	$P^I(r)$	$R^I(r)$	$P^F(r)$	$R^F(r)$	$f(r)$
0	0	0	1	0	1
1	1	0	0	0	0
2	1	1	0	1	4
3	0	1	0	2	2
4	0	2	1	1	3
5	1	2	1	2	5

Table 5.1: *Values of the pair* $(P(r), R(r))$ *described above for each rank and for both the initial and final partitioning of the example in Figure 5.3*

The problem – once again – is that as we are in a distributed setting, each rank r only knows its own $I(r), F(r)$ pairs. To be able to perform the reverse lookup described above, we need to add a synchronization phase in which every rank sends its F pair to all the other. This is called an All-Gather operation. At the end of this operation, each rank knows its own initial pair I as well as all the final pairs, allowing it to perform the search.

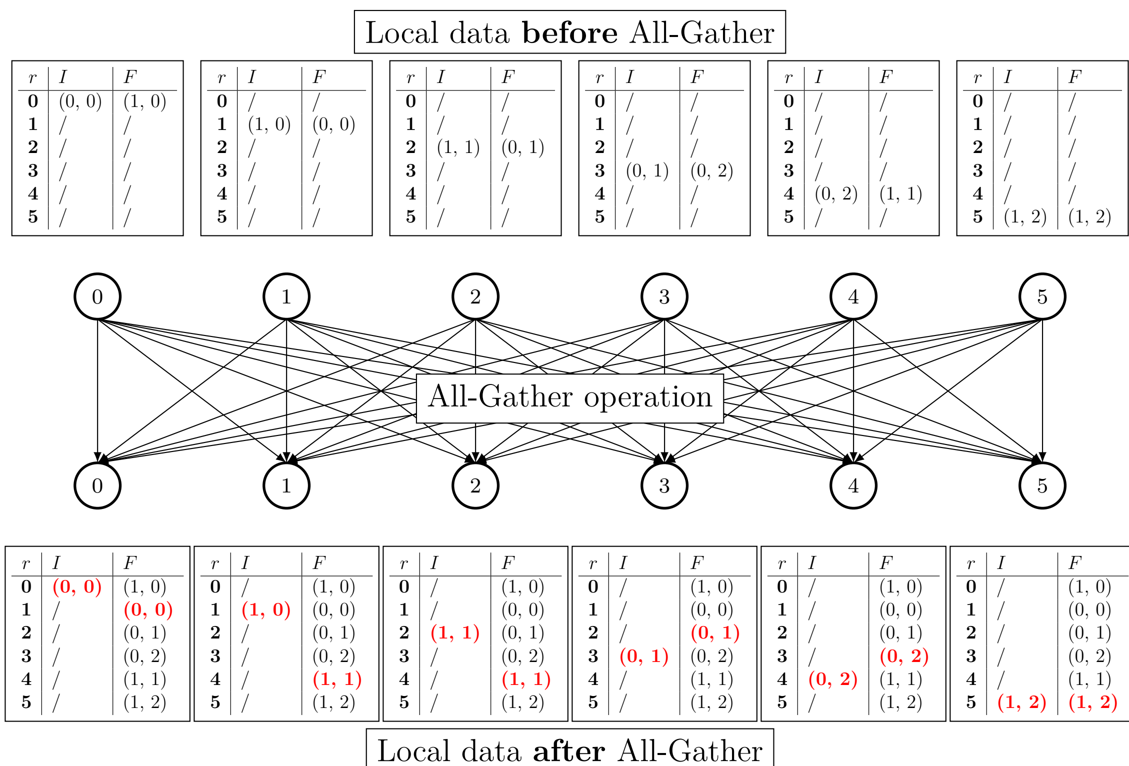


Figure 5.4: *Information available to all processes before/after All-Gather operation.*

Once each rank has found the value $r' = f(r)$, the partition-preserving isomorphism is applied by renaming each rank through the mapping. At the end of the operation, each rank r has been renamed to $f(r) = r'$ and the final partition is obtained.

5.1.2 Avoiding the All-Gather Operation

The All-Gather operation is lower-bounded by $\log_2(N)L + \frac{N-1}{N}\frac{m}{B}$ [Eijkhout et al., 2016] on its time cost, where we remind that N is the number of processes (equal to $|V|$), L and B are the latency and bandwidth of the system and m is the size of the messages being gathered. We show here that under a non-restrictive assumption on the initial allocation, we can avoid the All-Gather operation.

Assuming an initial block allocation with block size $b < |P|$ such that b is a divisor of $|P|$ and given any initial pair $I(r)$, we can compute the rank r it belongs to without synchronization. This actually means that we compute the inverse of the isomorphism.

To find the inverse isomorphism $f^{-1}(.)$, each rank r has to find a rank $r' : r = f^{-1}(r')$ such that $F(r) = I(r')$. We are now going in the reverse direction, searching for an initial pair matching a final pair. One could wonder where is the benefit in doing so. The

reason is that the method described earlier required a costly All-Gather operation whereas our new assumptions allow us to compute the value of $I(r)$ for every rank, **without synchronization**.

Assuming a block allocation with block size b and a k-way partition, the following formula finds the rank r' for which $F(r) = I(r')$.

$$r' = \left\lfloor \frac{R^F(r)}{b} \right\rfloor bk + P^F(r)b + (R^F(r) \bmod b)$$

Using this formula, each rank can compute the inverse mapping $r' : r = f^{-1}(r')$ without synchronization.

	$I(r)$		$F(r)$		Isomorphism	
r	$P^I(r)$	$R^I(r)$	$P^F(r)$	$R^F(r)$	$f^{-1}(r)$	$f(r)$
0	0	0	1	0	1	1
1	1	0	0	0	0	0
2	1	1	0	1	3	4
3	0	1	0	2	4	2
4	0	2	1	1	2	3
5	1	2	1	2	5	5

Table 5.2: *Update of Table 5.1 including the inverse of the partition-preserving isomorphism.*

We are still left with one problem: we don't actually know the isomorphism, only its inverse. Having the inverse is of no real use, as we need to rename the vertices according to the direct mapping. Fortunately, this step can be done cheaply as follows. Each rank r sends its ID to the rank r', meaning that every rank r' receives the value $r = f^{-1}(r') \implies r' = f(r)$. The mapping has been successfully inverted, each rank can now rename itself.

Chapter 6

Results

In this chapter, we analyze the added value of our partitioner.

We first compare its performance against the state-of-the-art open source implementation METIS [Karypis and Kumar, 1998]. For this comparison we look at both timet-to-partition and final objective value (edgecut). Furthermore, we make sure the partitioning respects the perfect balance constraint. For this analysis, we use communication graphs issued from the software FLUPS [Caprace et al., 2021] as well as theoretical 2D grids.

Afterward, we compare the time-to-solution of FLUPS with and without reordering in multiple test cases. This allows us to measure how qualitative the partitioning is.

6.1 Partitioner

In this section, we compare the time-to-partition and the edgecut of our partitioner described in Chapter 4 with results obtained with METIS. For this comparison, we use graphs obtained directly from FLUPS as well as theoretical two-dimensional grids. All of these graphs present a high degree of regularity. In the case of FLUPS, the communication graph is k-regular. This means every vertex of the graph has the same degree. Grids on the other hand are nearly k-regular. In 2D, the vertices internal to the grid have a degree of 4 whereas the vertices on the edges and corners have a degree of 3 and 2 respectively.

As the underlying algorithms are random, we perform 50 runs for each test case and take the average of the results. Additionally, unless explicitly specified, every test in this section is such that the resulting partition size is 128. Therefore, the number of partitions k is always set as $k = |V|/128$.

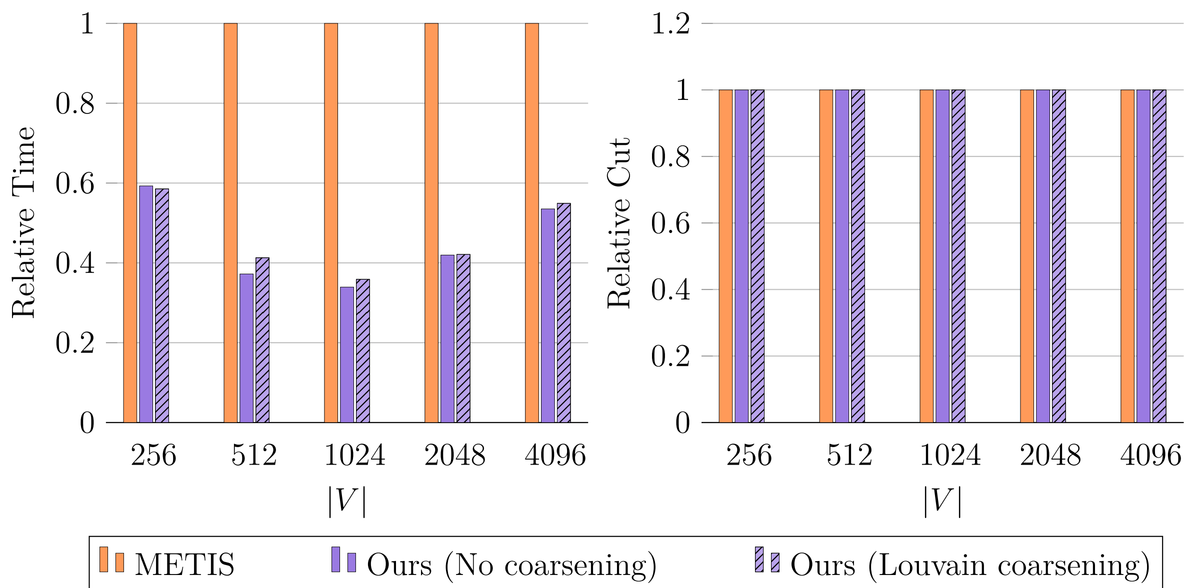


Figure 6.1: *Time-to-partition (left) and edgecut (right) of our partitioner with Louvain coarsening ( ) and without coarsening ( ) relative to METIS ( ) for communication graphs obtained from FLUPS.*

Figure 6.1 displays the timings and edgecuts of our partitioner relative to those of METIS when applied to FLUPS' communication graphs. **Both versions of our partitioner are faster than METIS for FLUPS' communication graphs and provide the same objective value.**

Except for $|V| = 256$ (2 partitions), we observe that our partitioner with Louvain coarsening ( ) is slightly slower than our partitioner without coarsening ( ). This is counterintuitive, as we explained in subsection 4.1.1 that the coarsening phase is supposed to mitigate the computational cost of partitioning by creating a hierarchy of smaller graphs. The reason is that the coarsening phase of the multilevel framework is only useful for very large graphs, for which the computational cost of the initial partitioning is prohibitive. For example, spectral partitioning has a computational cost of $\mathcal{O}(|V|^2)$ and is thus practically impossible to apply on a graph with millions of vertices. In our case the cost of coarsening actually outweighs the gains obtained during initial partitioning because the graphs are rather small. Furthermore, as our partitions are of constant size 128, we can actually only divide the size of our graph by this amount. Reducing the size by a factor greater than the partition's size would result in a failure of the partitioner, as the number of vertices of the smallest graph would be lesser than the required number of partitions. Additionally, we observe that both versions of our partitioner lead to the same quality provided by METIS, but in a maximum 60% of the time.

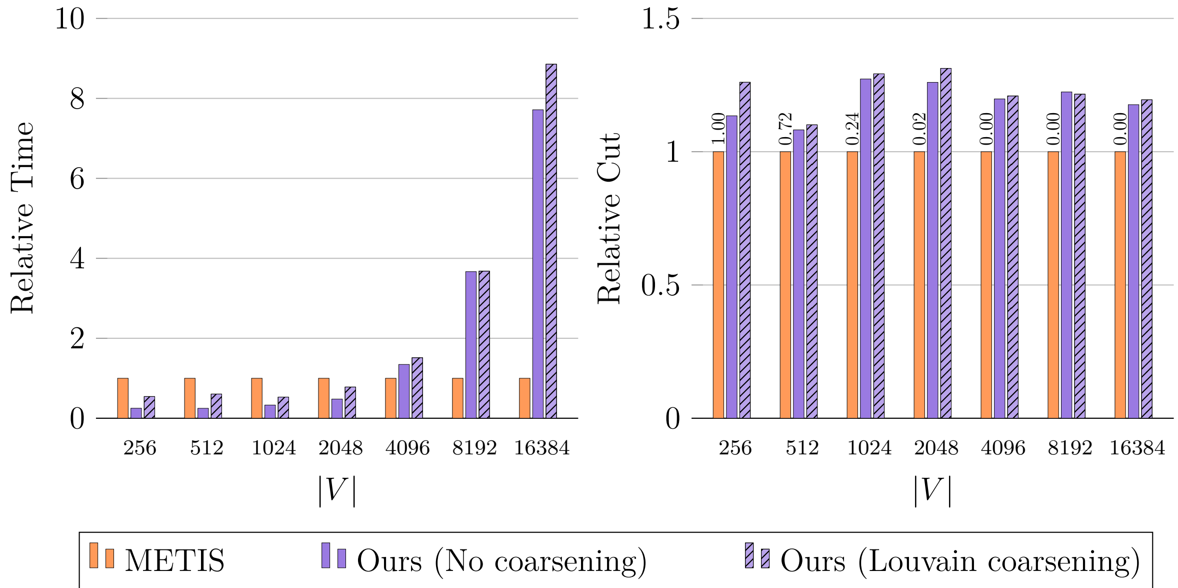


Figure 6.2: *Time-to-partition and edgecut of our partitioner with Louvain coarsening () and without coarsening () relative to METIS () for theoretical 2D grids. The number above METIS bars represents the fraction of samples respecting the perfect balance constraint.*

In the case of 2D grids, the results are noticeably different. Both of our partitioners are slower than METIS for $|V| \geq 4096$. In the case $|V| = 16384$, METIS is one order of magnitude faster than us. Furthermore, our partitioners are unable to retrieve the edgecut provided by METIS.

However, we find that METIS does not respect the balance constraint on every test case. Indeed, we observe that METIS leads to a perfect balancing everytime for $|V| = 256$, already decreasing to 72% of the time for $|V| = 512$. Starting at $|V| = 2048$, METIS does not lead to a perfect balancing sufficiently enough to be practically usable. Based on this observation, we conclude that METIS is not suitable for our use-case for $|V| \geq 4096$ as it does not respect the constraints of our problem. On the other hand, for $|V| \leq 2048$ there is a tradeoff between speed and accuracy. Indeed, our partitioner without coarsening () provides a 10-25% higher edgecut but in less than half the time of METIS. Furthermore, multiple trials have to be performed by METIS to ensure we find a perfectly balanced partition whereas our partitioner is guaranteed to respect the constraint, thus further increasing the time taken by METIS to find a suitable partitioning.

Additionally, we observe once again that applying the Louvain coarsening () seems to impact negatively both of the metrics of interests. Enabling Louvain coarsening leads to noticeably slower results, that is not compensated by a decrease in objective value.

In light of these results, the final partitioner used for reordering the ranks of the applications

discussed in section 6.2 will be our partitioner without coarsening step. Considering the observations made in this section, we believe this is the best course of action. Indeed, we have first observed that our partitioner without coarsening provided faster results then METIS on FLUPS' graphs while still resulting in the same objective value. On the other hand, we have observed cases where METIS failed to respect the constraints of our problem.

6.2 Reordering

In this section, we analyze the added value of rank-reordering on real MPI applications. For this purpose, our metric of interest is the *time-to-solution*. This corresponds to the time taken by the application to compute one solution to its problem.

6.2.1 The Systems

For completeness, let us first describe the two systems used to perform the tests of this section.

Property	MeluXina	Lucia
Processor type	2x AMD EPYC Rome 7H12 (64c)	2x AMD EPYC 7763 (64c)
Clock speed	**2.6GHz**	**2.45GHz**
Cores per Node	128	128
Memory (RAM)	512 GB	256 GB
Interconnect	**1x HDR200 (200Gb/s)**	**1x HDR100 (100Gb/s)**
Topology	Dragonfly+	Fat Tree
MPICH version	4.2.0 (source)	4.2.0 (source)
UCX version	1.12.1 (EasyBuild)	1.13.1 (EasyBuild)

Table 6.1: *Cluster-node properties of both MeluXina and Lucia supercomputers. Note that memory is given in giga-BYTE (GB) and bandwidth is expressed in giga-BIT (Gb), therefore the theoretical bandwidth are 25GB/s and 12.5GB/s.*

Table 6.1 displays the properties of a single cluster-node on each of the MeluXina and Lucia systems. The two important differences are the clock speed of the processors and the speed of the network. In both categories, MeluXina offers a better product than Lucia. Furthermore, we remind the reader of the results of Figure 3.2 discussed in Chapter 3 in which we observed that MeluXina displayed a similar local and network bandwidth. Lucia's network bandwidth on the other hand is twice slower than its local data transfer. Following these observations, it is natural to assume MeluXina will perform better on every test cases. Finally, we used the FLUPS commit 4423a4b compiled from source

using GCC 11.3, available on both systems. Both MPICH and FLUPS were compiled in optimized mode.

6.2.2 Proof of concept

We first apply our rank-reordering library to a simple example serving two purposes. First, it validates the idea that reordering can lead to speedup. Secondly, as we do not perform any computations it enables us to view the effects of reordering *on communications*. Of course, the computations speed should not be impacted.

In this section, we will often mention the term *speedup*. Although this term has a clear definition in HPC, that is the ratio of the time taken by a sequential algorithm and a parallel algorithm solving the same task (i.e. T_{seq}/T_{par}), we use this term rather broadly to express the ratio of time-to-solution between two cases. As our focus is on rank-reordering, we generally use the term speedup to express the ratio of non-reordered and reordered examples.

Low Number of Big Messages

The example application here is represented in Figure 6.3 and works as follows:

1. Reserve two cluster-nodes with N rank per nodes (2xN ranks in total). The first node receives ranks $\{0, 1, ..., N-1\}$ and the second receives $\{N, N+1, ..., 2N-1\}$, this is a *linear* allocation.

2. Each rank $r \in \{0, 1, ..., 2N-1\}$ reserves a buffer of size 4MB and initialize a receive requests on this buffer.

3. Each rank $s \in \{0, 1, ..., N-1\}$ on the first node sends a non-blocking message of size 4MB to the rank $r = s + N \in \{N, ..., 2N-1\}$.

4. Each rank $r = s + N \in \{N, ..., 2N-1\}$ on the second node sends a non-blocking message of size 4MB to the rank $s = r - N \in \{0, 1, ..., N-1\}$.

We chose the message size (4MB) to fit with the highest size displayed in the OSU-benchmarks (Figure 3.2 and Appendix B). We measure the time-to-solution of this application as the time elapsed between the start of step (3) and the end of step (4) after a few rounds of warm up. As each rank performs a local measure, we take the global measure as the average of all the local measures. Notice that contrarily to the results of the OSU-Benchmarks, we use *non-blocking communications* to better fit the inner working of FLUPS, discussed in the next section. As a reminder, a non-blocking communication is a kind of communication that does not use implicit synchronization (i.e. the receiver does not wait for the message to arrive). Rather, the user initializes the request of the

message and then choose when to wait for the reception. This kind of communications allows performing latency-hiding, in which you perform some computations in between the initialization of the request and the reception of the message.

We look at the behavior of this application with regard to the number of rank per cluster-node. In practice, this kind of test is called a weak-scaling. The quantity of work per rank is the same regardless of the number of rank hence the total quantity of work increases proportionally with the number of ranks. In a perfect case, we hope that the time-to-solution remains constant regardless of the number of ranks.

We can see on the visual representation of Figure 6.3 that in this simple application the initial allocation causes all communications to pass through the network. Fortunately it has a simple optimal solution with an objective value of 0, meaning that all communications on the reordered ranks happens locally to the cluster-nodes.

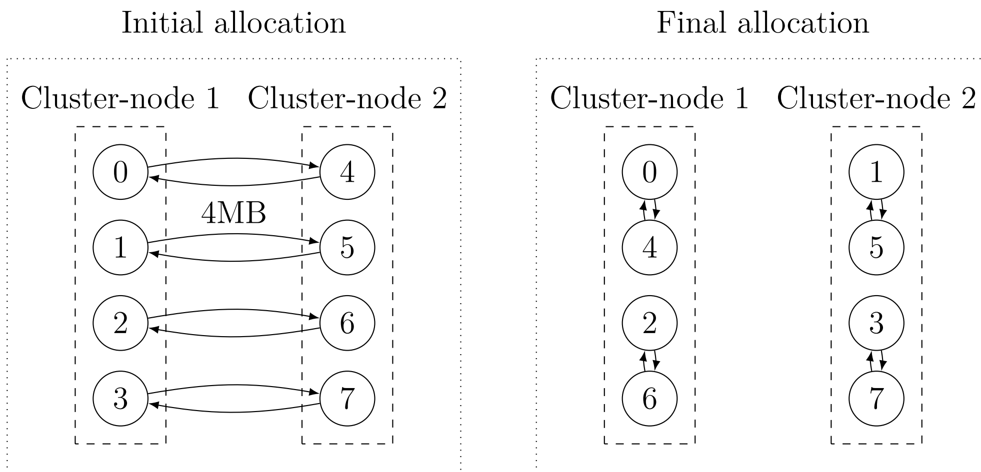


Figure 6.3: *Representation of the proof-of-concept user application with $N = 8$ ranks. (Left) Suboptimal initial linear allocation. (Right) Final reordered allocation.*

Of course, an advised user would have asked the resource manager to enforce a round-robin[1] initial allocation instead of a linear one. This example is to show that the user does not need to provide additional information to recover an efficient process allocation. In this example, the optimal allocation is trivial and can be enforced by the user on any process launcher without much trouble. But in more complex applications the user might not know the optimal rank ordering, nor even a decent one.

The results for this example application are shown in Figure 6.4. On which we observe that rank reordering leads to speedups. On Lucia, we clearly distinguish two different

[1]This is the technical name of the allocation displayed in the right of Figure 6.3. In this allocation, the resource manager distributes one rank to every node and then cycle back to the first node until all ranks have been allocated

regimes. The first is between 8 and 32 ranks with a speedup of about 2.5, the second is between 48 and 128 with a speedup between 5 and 8. The presence of these two regimes cannot be explained from the time cost model described in Chapter 3. We assume that this behavior is due to the network interface controller (NIC). This piece of hardware is responsible for handling the communication requests and sending/receiving data to and from the network. Like all hardware, it has physical limitations on the number of requests it can handle concurrently and the speed at which it can treat those requests. We believe this small example reaches Lucia's NIC threshold of maximal concurrent requests when $N \geq 48$. This behavior is not displayed on MeluXina, which shows an approximate speedup of 1.8 across all N.

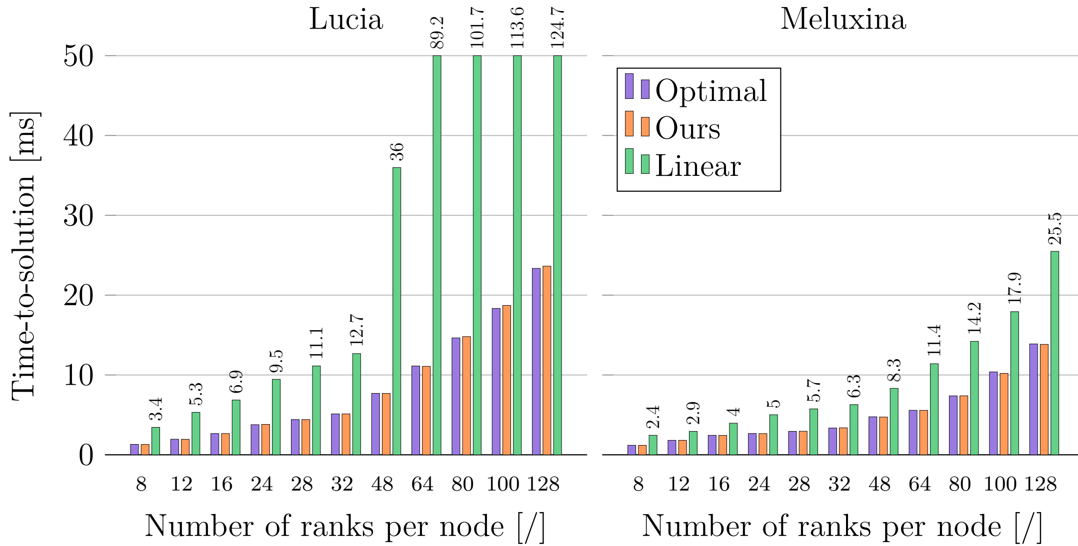


Figure 6.4: *Time-to-solution of the example user app described by Figure 6.3. We display the optimal allocation (▮▮), with no network communications. Linear allocation (▮▮) with every communication occurring on network. Ours (▮▮) is the linear allocation to which we applied our rank reordering algorithm. For clarity, we added the value of the linear allocation above its bars.*

The constant speedup of 1.8 for MeluXina hints that its NIC is capable of handling more concurrent requests. Furthemore, this speedup cannot be explained by our model. As observed in Figure 3.2, MeluXina has approximately equal network and local latency/bandwidth for heavy messages. Meaning that we should not actually observe a speedup on MeluXina. **This leads us to believe there is a network behavior we did not take into account in the modelling phase.** This assumption is reinforced by the discrepancies of the optimal (▮▮) and reordered (▮▮) time-to-solution between Lucia and MeluXina. Once again, both systems display a similar latency/bandwidth for large messages and therefore we shouldn't observe a 60% difference between the two systems.

However, the increased gains on Lucia can easily be explained by the fact that it has only half the network bandwidth of MeluXina.

Regardless of the system and the number of ranks, we observe for this example that reordering ( ) yields a time-to-solution equivalent to the optimal allocation ( ). For this example, our algorithm is capable of retrieving the optimal allocation without any additional user-provided information.

High Number of Small Messages

We now perform a slight modification of the example used above. In the example above, we sent a few heavy messages. As we have seen, MeluXina enjoying twice as much network bandwidth as Lucia displayed lower speedups for large messages. We modify the application such that each rank sends 40.000 messages of 4 bytes (one integer) each. Note that the messages are sent in a "ping-pong" manner. For example, rank 0 send a 4 bytes message to rank N, that will then send a 4 bytes message in response. This cycle is repeated 40.000 times, and we measure the time from the first message to the last.

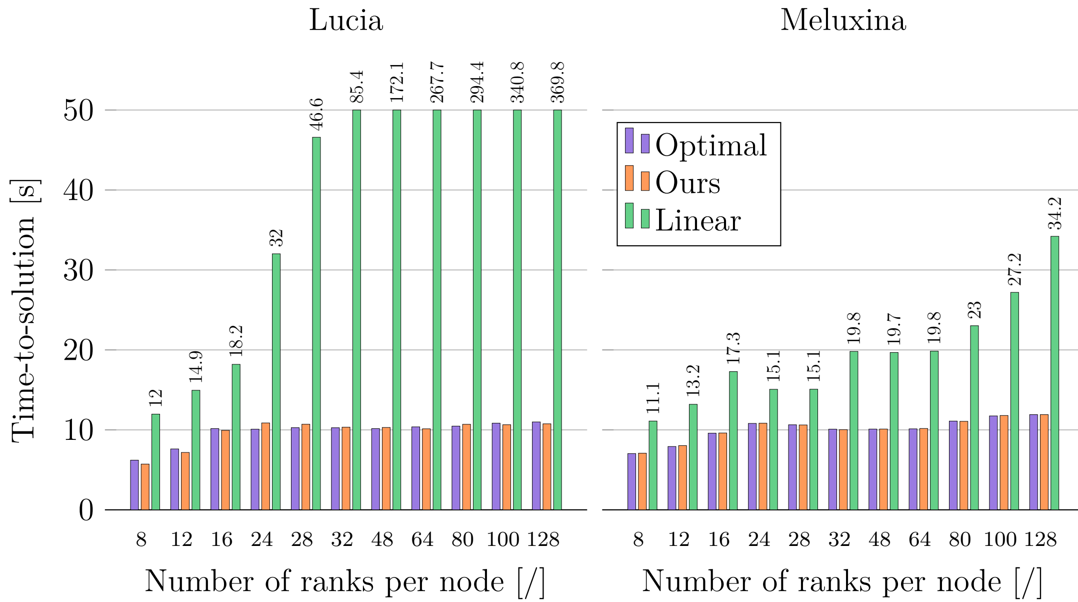


Figure 6.5: *Time-to-solution of the* ***modified*** *example user app. We display the optimal allocation ( ) with no network communications. Linear allocation ( ) with every communication occurring on network. Ours ( ) is the linear allocation to which we applied our rank reordering algorithm. For clarity, we added the value of the linear allocation above its bars.*

In this case, shown in Figure 6.5, there is a large discrepancy between the behavior of the

linear allocation (▮▮) on the two systems. Lucia displays an impressive speedup going from 1.79 to 33.64 whereas MeluXina's speedup stays between 1.39 and 2.87. Once again, this kind of behavior is not predicted by our extended latency/bandwidth model. We believe the poor results of the linear allocation (▮▮) on Lucia are due to the physical limits of the NIC. This assumption is corroborated by the behavior of the linear allocation on Lucia:

$N \leq 16$ We observe an approximately constant speedup, which would mean that the NIC is operating under its physical limits.

$24 \leq N \leq 48$ We observe a transition phase with a sharp rise in speedup, this would correspond to the NIC starting to reach its physical limits.

$64 \leq N$ The speedup increases linearly with the number of ranks. We assume that this behavior is due to the NIC being flooded with requests, causing messages to wait before being enqueued. As the ranks each send a constant number of messages, the number of messages waiting to be enqueued grows linearly with the number of ranks.

On MeluXina, we do not observe the sharp rise in speedup. However, the speedup also seems to increase linearly starting from 64 ranks.
Notice that compared with the previous example of Figure 6.4 both Lucia and MeluXina display the same time-to-solution for the optimal (▮▮) and reordered (▮▮) cases.

Once again however, our reordering (▮▮) is able to retrieve a time-to-solution equivalent to the optimal solution.

6.2.3 FLUPS

As stated earlier, we use FLUPS as our application of interest. FLUPS is a distributed fast-fourier-transform used to solve Poisson's equation in 3D with a variety of boundary conditions. We explain briefly the application communication pattern here. The reader can refer to [Caprace et al., 2021, Balty et al., 2023] for more detailed information.

Topology Switches

Initially, FLUPS receives a 3D rectangular physical domain of length L_x, L_y, L_z in each direction. We assume that this global domain is already decomposed in blocs and distributed among the ranks. For this decomposition, we assume we have N ranks forming a cartesian grid topology with $P_x \times P_y \times P_z$ ranks along the edges. With such a decomposition, each rank is responsible for a subdomain of size $L_x/P_x \times L_y/P_y \times L_z/P_z$.

To be able to perform the one dimensional FFTs in each direction, FLUPS needs to exchange subdomains such that each rank has all the data of the physical domain in

the FFT's direction, this is called a pencil decomposition. A 2D example is shown in Figure 6.6. In this figure, we see the initial block decomposition in 16 subdomains. To perform the FFTs in the x direction, each rank needs to exchange part of its data with all the ranks in its row. For the second FFT, each rank sends data to the ranks in its column. In a 3D case, the same principle applies, and we would have an extra exchange in the z direction. Each exchange from one direction to another is called a *topology switch*. These topology switches will be our main focus as they are the sources of communications.

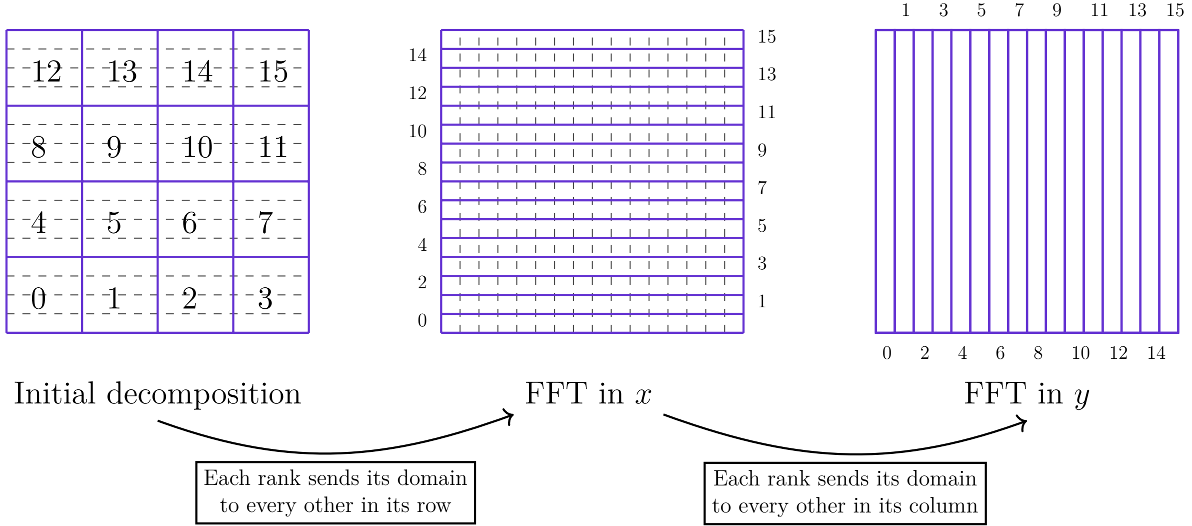


Figure 6.6: *Example of FLUPS's topology switches in a 2D case with 16 ranks. The solid lines represent the data local to each rank whereas the dashed lines represent the chunks of data that need to be exchanged during the topology switch.*

Notice that the size of the messages is not constant across the topology switches. Indeed, let us assume that each rank in the example of Figure 6.6 is responsible for a subdomain of unit size. The initial decomposition is a 4×4 grid, that we will simply note as $(4, 4)$. To perform the first topology switch, we go from a $(4, 4)$ to a $(1, 16)$ decomposition, meaning each rank sends 3 messages of size $\frac{1}{4}$ (the rank does not send data to itself). The second topology switch goes from a $(1, 16)$ to a $(16, 1)$ decomposition, now each rank sends 15 messages of size $\frac{1}{16}$.

One last consideration is that for the unbounded boundary conditions we will use, FLUPS performs a technique called "domain doubling". The details of this technique are not relevant to us, the only important information is that it doubles the size of the domain at each topology switch. Therefore, if we set the total domain size as 1, the size of the domain for each topology switch would be 1 for x, 2 for y and 4 for z.

FLUPS Reordering

We now apply our rank-reordering library to FLUPS and compare the results on both systems. Our test case is a 3-dimensional fully unbounded domain. As we are interested in the communication times of the application, we perform a weak-scaling. Each rank will thus receive a local domain of constant size, meaning the global domain size will grow proportionally to the number of ranks.

We based our tests to be equivalent to the ones performed in [Balty et al., 2023]. The domain decomposition used here is given by Table 6.2. Using this processor decomposition, we test two cases with 64^3 and 96^3 points per rank. The size of the global domain is then respectively $(64P_x)(64P_y)(64P_z)$ and $(96P_x)(96P_y)(96P_z)$.

Nodes	P_x	P_y	P_z	Topo X	Topo Y	Topo Z
1	4	4	8	(1, 16, 8)	(16, 1, 8)	(16, 8, 1)
2	4	8	8	(1, 32, 8)	(32, 1, 8)	(32, 8, 1)
4	8	8	8	(1, 64, 8)	(64, 1, 8)	(64, 8, 1)
8	8	8	16	(1, 64, 16)	(64, 1, 16)	(64, 16, 1)
16	8	16	16	(1, 128, 16)	(128, 1, 16)	(128, 16, 1)
32	16	16	16	(1, 256, 16)	(256, 1, 16)	(256, 16, 1)
64	16	16	32	(1, 256, 32)	(256, 1, 32)	(256, 32, 1)
128	16	32	32	(1, 512, 32)	(512, 1, 32)	(512, 32, 1)

Table 6.2: *Domain decomposition with regard to the number of cluster-nodes. We also show the successive decompositions for each topology switch.*

Figure 6.7 shows the time-to-solution of FLUPS with 64^3 unknowns per rank which corresponds to approximately $2MB$ of data per rank.

For a single cluster-node – and thus *without network communications* – we can make two observations. First, Lucia is slower than MeluXina by approximately 20%. We have seen in Table 6.1 that MeluXina had a 6% faster CPU clock speed than Lucia. Furthermore, for a single node the topology switch from the initial decomposition to the x topology requires sending 3 messages of size $\frac{1}{2}MB$. Then the x to y switch requires sending 15 messages of size $\frac{1}{8}MB$. After doubling the size of the domain, the y to z switch requires sending 7 messages of size $\frac{1}{2}MB$. For those two sizes, the OSU-Benchmarks shows that MeluXina provides a respectively 21% and 13% faster local bandwidth with similar latency. Computing a weighted average over the number of messages leads to an average of 18.5% faster communications on MeluXina. Coupling this with the increased clock speed of the CPU leads to a value close to the observed 20% difference. The second observation is that reordering does not lead to any speedup on a single node. This was expected, as no communication occurs on the network, we do not actually reorder the ranks.

We now look at the results on multiple nodes. First, we observe that reordering is detrimental to the time-to-solution on both systems when run on 32 and 128 nodes but beneficial for all other configurations. We can actually see that the rank-reordering favored the communications of the 3rd topology switch on all configurations. This is understandable, given the fact that the domain doubling technique doubles the cost of communications after each switch, the 3rd switch is the most costly. This is clearly observable on the non-reordered case of Figure 6.7 in which all communications due to a topology switch takes more time than the previous one.

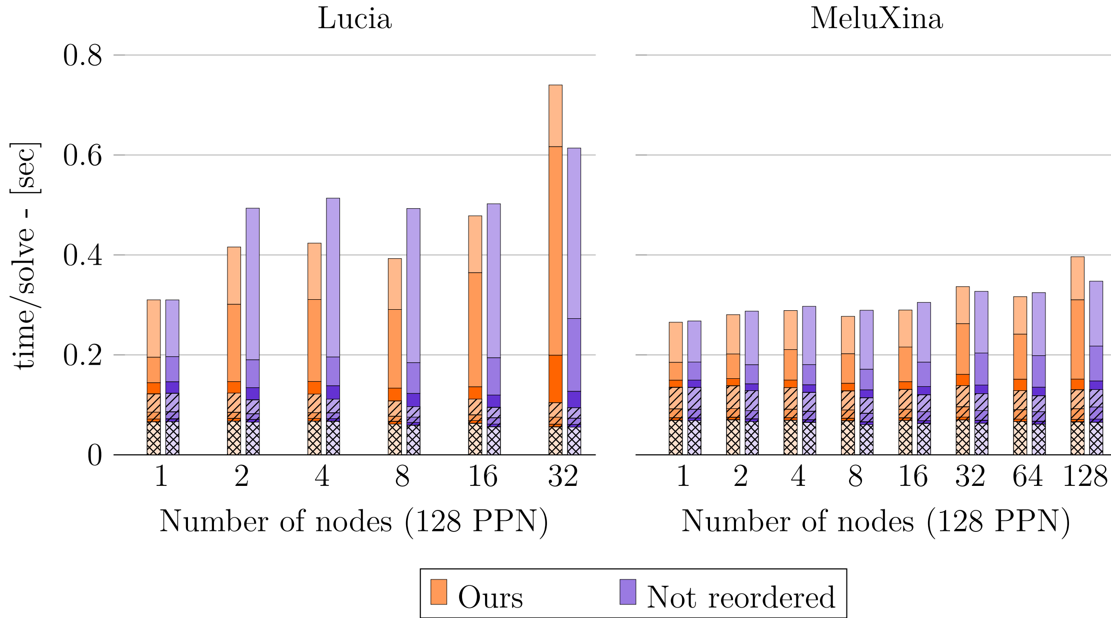


Figure 6.7: *Time-to-solution of FLUPS with and without reordering on up to 128 ranks for both Lucia and MeluXina. We use* 64^3 *data points per rank in a fully unbounded domain. Cross-hashed bars are the FFTs, hashed bars represent the time spent computing after initiating the exchange requests, plain bars represent waiting time. The shades are the different topology switch.*

Looking at the reordered case, we can see that the 3rd topology switch has benefitted from a 3 time speedup on Lucia and a 25% speedup on MeluXina across all configurations. However, there is no free lunch. The cost of this decrease in communication time is an increase in the time spent in the second topology switch. On Lucia, the increase in the second topology switch in also 300% but except for the 32 nodes configuration the gains on the third topology outweighs the loss in the second. On MeluXina however, the increase in the second topology switch is about 40%, except for the 128 nodes configuration where it is higher, with a 100% increase.

Overall, the effects of reordering are clearly visible on Lucia and provide a

small benefit on MeluXina. However, for some configurations the gains in the communication time of the 3rd topology switch are not enough to compensate for the losses in the second one. We believe this behavior comes from a deficiency in the weighting of the edges.
Indeed, the two detrimental cases have the following topology switches:

$$32 \text{ nodes} : (1, 256, 16) \rightarrow (256, 1, 16) \rightarrow (256, 16, 1)$$
$$128 \text{ nodes} : (1, 512, 32) \rightarrow (512, 1, 32) \rightarrow (512, 32, 1)$$

Looking at the values of Table 6.2, these are the only two cases of the form $(1, 16\alpha, \alpha) \rightarrow (16\alpha, 1, \alpha) \rightarrow (16\alpha, \alpha, 1)$. This means the number of messages required to change from the x to the y topology is more than 16 times greater than what is needed to change from the y to the z topology. This behaviour is a hint towards a deficiency in the latency part of the edge weighting. Indeed, our algorithm chose to favor the small number of heavy communications of the 3rd topology switch over the large number of light communications of the 2nd switch. After comparing both, we have observed on Figure 6.7 that the large number of light communications were actually more time-consuming, therefore leading us to believe the latency term of our edge weighting is not well-balanced with the bandwidth term.

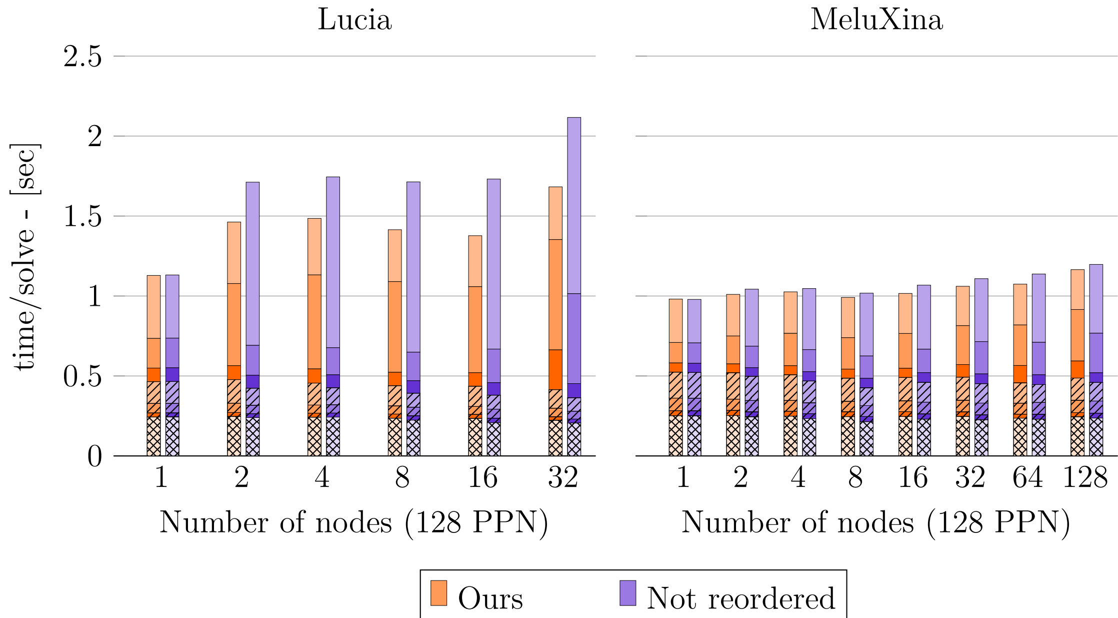


Figure 6.8: *Time-to-solution of FLUPS with and without reordering on up to 128 ranks for both Lucia and MeluXina. We use* 96^3 *data points per rank in a fully unbounded domain. Cross-hashed bars are the FFTs, hashed bars represent the time spent computing after initiating the exchange requests, plain bars represent waiting time. The shades are the different topology switch.*

We now test our algorithm on a larger domain with 96^3 data points per rank. The results are shown in Figure 6.8. In this case, the reordering is beneficial for all configurations on both systems, further increasing our belief that the two outlier cases discussed above are due to a deficiency in our latency term.

Once again, the effects of reordering are more pronounced on Lucia due to its weaker network. On Lucia, the reordering provides a speedup of approximately 15-20% depending on the configuration whereas on MeluXina the speedup is only about 2-3%. We can see on this case that the reordering made the same choice of optimizing the communications for the last topology switch. This results in a notable decrease in communication time for the 3rd switch while at the same time increasing the cost of the second switch.

With this reordering technique, a job previously running for 2 days on 32 nodes of Lucia could now be completed in 1 day and 14 hours. This represents a saving of 10 hours of human time and 40960 core-hours. Assuming a cost of 1cent/core-hour, this represents approximately 400 euros of savings for this job.

Chapter 7

Conclusion

The results obtained during this thesis indicate that rank-reordering techniques computed using graph-partitioning are able to reduce the communication costs of distributed MPI applications. We have shown that rank-reordering requires to compute a special kind of graph isomorphism – that we named *partition-preserving* – between the initial allocation provided by the resource manager of the supercomputer and the final allocation given by the graph-partitioner based on the communication patterns of the application. Our contribution in that direction is a user-friendly library able to automatically reorder the rank of an MPI application solely based on the message list of every rank. This library has been tested with FLUPS and we have shown that it is able to reduce the overall time-to-solution by up to 20% on Lucia. The effects were not that impressive on MeluXina, showing reordering is most important on systems having lower bandwidth.

Despite these good results, there remains a clear limitation. As we only considered the case where the number of process per node of the supercomputer is constant, our library is not able to tackle a variable number of processes per node. This limitation arises when using shared nodes, in which users share parts of the nodes of a supercomputer.

We see multiple directions for further research. The first one would be to include the behavior of the network interface controller (NIC) into the modelling of the cost, this would enable a better understanding of the network congestion resulting from the application. A second research direction – that we find particularly interesting – would be to apply some of these techniques to *crowd computing*. In this branch of distributed computing, the ranks are not distributed among the processors of a supercomputer, linked with specialized high bandwidth cables, but rather among processors of entirely different machines that could be located kilometers away from each other, linked only through the internet. We believe such reordering techniques would be particularly useful for this field, but this area of research would first require solving the limitation discussed above as the machines could very well have entirely different properties.

Appendix A

Why Directed Graphs Are Inefficient ?

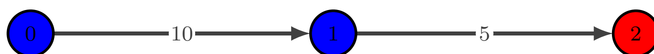
The size of our graphs forces us to use a compressed sparse row (CSR) representation of the adjacency matrix. This representation consists of 3 vectors

- I is the cumulative number of neighbors of the vertices such that $[I_u, I_{u+1}]$ is the window of indices where we find information about the neighbors of u in J and W.
- J is the ID of the neighboring vertex. J_{I_u+k} gives the k-th neighbor of u, with $I_u + k < I_{u+1}$.
- W gives the corresponding edge weight.

This representation allows to efficiently query the neighbors of a given vertex (e.g. given u, explore all its neighbors). On the other hand, it does not allow reversing the neighboring relation efficiently (e.g. given v, is v a neighbor of u) because we do not have direct access to columns.

At several steps of the algorithms described in this thesis, we had to compute the gain $g_u(P_j)$ of moving vertex u from its partition $P(u)$ to the partition P_j. This gain is defined as the difference between the weight of the edges linking u to its new partition P_j and the weight of the edges linking u to its old partition $P(u)$.

$$g_u(P_j) = E_j(u) - I(u) = \sum_{v \in P_j} w_{uv} - \sum_{v \in P_{P(u)}} w_{uv}$$



CSR representation : $I = \begin{bmatrix} 0 & 1 & 2 \end{bmatrix}$ $J = \begin{bmatrix} 1 & 2 \end{bmatrix}$ $W = \begin{bmatrix} 10 & 5 \end{bmatrix}$

In the example above, we compute $g_1(red) = 5$. We would then be tempted to move vertex 1 to the red partition, which is clearly a bad idea. The problem is that to compute the true gain of -5, we would need to explore the neighbors of every other vertex to make sure vertex 1 is not a neighbor to them. That would mean we have to visit every edge of the graph to compute a single cost, which is clearly not efficient.

This problem does not appear in undirected graphs because the neighboring relation is symmetric.

Appendix B

OSU Benchmarks [MVA, 2024]

	Meluxina		Lucia	
Size	latency	Bandwidth	latency	Bandwidth
2^2	1.11	13.87	1.43	10.51
2^3	1.10	28.92	1.43	21.02
2^4	1.11	57.96	1.43	41.99
2^5	1.21	115.12	1.54	83.38
2^6	1.32	220.43	1.59	192.70
2^7	1.35	429.05	1.64	394.30
2^8	1.85	792.11	1.96	740.56
2^9	1.99	1377.22	2.10	1436.10
2^{10}	2.34	2588.10	2.55	2690.21
2^{11}	2.37	4410.15	2.69	4633.97
2^{12}	2.87	6707.33	3.35	7690.05
2^{13}	3.34	9552.95	4.10	10481.91
2^{14}	5.56	18036.46	5.72	11310.21
2^{15}	6.55	20739.38	7.10	11761.30
2^{16}	8.25	22032.73	10.53	12075.25
2^{17}	10.94	22605.51	15.85	12212.19
2^{18}	16.41	22817.18	26.47	12278.17
2^{19}	27.57	22834.29	47.78	12311.00
2^{20}	50.31	23520.43	91.14	12328.53
2^{21}	95.69	24658.79	176.53	12336.37
2^{22}	181.18	24405.77	346.21	12340.34

Table B.1: *Results of the OSU-benchmarks* `osu_bw` *and* `osu_lat` *on network communication for both MeluXina and Lucia systems. Message size is expressed in bytes, latency in μs and bandwidth in MB/s.*

	Meluxina		Lucia	
Size	latency	Bandwidth	latency	Bandwidth
4	0.16	50.56	0.52	28.21
8	0.18	103.56	0.51	58.14
16	0.16	209.30	0.51	113.89
32	0.16	389.93	0.71	198.00
64	0.16	851.62	0.69	423.46
128	0.21	1328.86	0.76	440.76
256	0.21	2628.47	0.76	825.65
512	0.23	5058.91	0.85	1603.67
1024	0.27	9397.88	1.02	2864.21
2048	0.32	12956.23	1.60	2746.17
4096	0.41	17579.07	2.21	3303.86
8192	0.58	23267.79	3.33	3465.48
16384	4.13	9607.99	4.21	9723.81
32768	4.88	16835.34	5.00	14670.09
65536	6.32	24603.76	6.41	18623.96
131072	9.14	26655.01	9.30	21958.82
262144	15.50	26825.92	14.72	23461.66
524288	28.45	26575.41	26.11	23595.78
1048576	46.59	25739.14	48.15	23645.31
2097152	87.54	24848.10	92.24	23608.95
4194304	171.20	24798.50	187.75	23659.09

Table B.2: *Results of the OSU-benchmarks* `osu_bw` *and* `osu_lat` *on local communication for both MeluXina and Lucia systems. Message size is expressed in bytes, latency in μs and bandwidth in MB/s.*

Bibliography

[MVA, 2024] (2024). Osu micro-benchmarks.

[Abbas et al., 2018] Abbas, Z., Kalavri, V., Carbone, P., and Vlassov, V. (2018). Streaming graph partitioning: an experimental study. *Proc. VLDB Endow.*, 11(11):1590–1603.

[Balty et al., 2023] Balty, P., Chatelain, P., and Gillis, T. (2023). FLUPS – a flexible and performant massively parallel Fourier transform library. *IEEE Transactions on Parallel and Distributed Systems*, 34(7):2011–2024. arXiv:2211.07777 [physics].

[Blondel et al., 2008] Blondel, V. D., Guillaume, J.-L., Lambiotte, R., and Lefebvre, E. (2008). Fast unfolding of communities in large networks. *Journal of Statistical Mechanics: Theory and Experiment*, 2008(10):P10008. arXiv:0803.0476 [cond-mat, physics:physics].

[Caprace et al., 2021] Caprace, D.-G., Gillis, T., and Chatelain, P. (2021). FLUPS: a Fourier-based Library of Unbounded Poisson Solvers. *SIAM Journal on Scientific Computing*, 43(1):C31–C60. arXiv:2006.09300 [physics].

[Carvalho, 2002] Carvalho, C. (2002). The gap between processor and memory speeds.

[Drepper, 2007] Drepper, U. (2007). What every programmer should know about memory.

[Eijkhout et al., 2016] Eijkhout, V., van de Geijn, R., and Chow, E. (2016). *Introduction to High Performance Scientific Computing.*

[Fiduccia and Mattheyses, 1982] Fiduccia, C. and Mattheyses, R. (1982). A Linear-Time Heuristic for Improving Network Partitions . In *19th Design Automation Conference*, pages 175–181. ISSN: 0146-7123.

[Fiedler, 1975] Fiedler, M. (1975). A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak mathematical journal*, 25(4):619–633.

[Garey et al., 1976] Garey, M. R., Johnson, D. S., and Stockmeyer, L. (1976). Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3):237–267.

[George, 1973] George, A. (1973). Nested dissection of a regular finite element mesh. *SIAM Journal on Numerical Analysis*, 10(2):345–363.

[Gilbert et al., 1998] Gilbert, J. R., Miller, G. L., and Teng, S.-H. (1998). Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110.

[Hendrickson and Leland, 1995a] Hendrickson, B. and Leland, R. (1995a). An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469.

[Hendrickson and Leland, 1995b] Hendrickson, B. and Leland, R. (1995b). A multi-level algorithm for partitioning graphs. page 28–28.

[Hu et al., 1999] Hu, Y., Lu, H., Cox, A., and Zwaenepoel, W. (1999). Openmp for networks of smps. page 302.

[Karypis and Kumar, 1998] Karypis, G. and Kumar, V. (1998). A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392. Publisher: Society for Industrial and Applied Mathematics.

[Karypis and Kumar, 1999] Karypis, G. and Kumar, V. (1999). Kumar, V.: A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. SIAM Journal on Scientific Computing 20(1), 359-392. *Siam Journal on Scientific Computing*, 20.

[Kernighan and Lin, 1970] Kernighan, B. W. and Lin, S. (1970). An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307. Conference Name: The Bell System Technical Journal.

[Lasalle et al., 2015] Lasalle, D., Mostofa, M., Patwary, A., Satish, N., Sundaram, N., Karypis, G., and Dubey, P. (2015). Improving Graph Partitioning for Modern Graphs and Architectures.

[Message Passing Interface Forum, 2023] Message Passing Interface Forum (2023). *MPI: A Message-Passing Interface Standard Version 4.1.*

[Meyerhenke et al., 2014] Meyerhenke, H., Sanders, P., and Schulz, C. (2014). Partitioning Complex Networks via Size-constrained Clustering. arXiv:1402.3281 [cs].

[Mislove et al., 2007] Mislove, A., Marcon, M., Gummadi, K. P., Druschel, P., and Bhattacharjee, B. (2007). Measurement and analysis of online social networks. In *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, IMC '07, page 29–42, New York, NY, USA. Association for Computing Machinery.

[Newman, 2004] Newman, M. E. J. (2004). Fast algorithm for detecting community structure in networks. *Physical Review E*, 69(6):066133. arXiv:cond-mat/0309508.

[Newman, 2006] Newman, M. E. J. (2006). Modularity and community structure in networks. *Proceedings of the National Academy of Sciences*, 103(23):8577–8582.

[Predari and Esnard, 2016] Predari, M. and Esnard, A. (2016). A k-way Greedy Graph Partitioning with Initial Fixed Vertices for Parallel Applications. page 8.

[Sanders and Schulz, 2012] Sanders, P. and Schulz, C. (2012). Think Locally, Act Globally: Perfectly Balanced Graph Partitioning. arXiv:1210.0477 [cs].

[Schulz et al., 2019] Schulz, C., Träff, J. L., and von Kirchbach, K. (2019). Better Process Mapping and Sparse Quadratic Assignment. arXiv:1702.04164 [cs].

[Strohmaier et al., 2023] Strohmaier, E., Dongarra, J., Simon, H., and Meuer, M. (2023). Top 500 supercomputers : November 2023.

[Traag et al., 2019] Traag, V. A., Waltman, L., and van Eck, N. J. (2019). From Louvain to Leiden: guaranteeing well-connected communities | Scientific Reports. *Scientific Reports*, 9(1):5233.

[Wang et al., 2023] Wang, C., Guo, Y., Balaji, P., and Snir, M. (2023). Near-lossless mpi tracing and proxy application autogeneration. *IEEE Transactions on Parallel and Distributed Systems*, 34(1):123–140.

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**