TU WIEN Informatics

Efficient Process Mapping for Cartesian Topologies

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Markus Lehr
Matrikelnummer 01426019

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Jesper Larsson Träff

Wien, 2. Dezember 2019

Markus Lehr　　　Jesper Larsson Träff

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Informatics

Efficient Process Mapping for Cartesian Topologies

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Markus Lehr
Registration Number 01426019

to the Faculty of Informatics

at the TU Wien

Advisor: Jesper Larsson Träff

Vienna, 2nd December, 2019

Markus Lehr	Jesper Larsson Träff

Erklärung zur Verfassung der Arbeit

Markus Lehr

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 2. Dezember 2019

Markus Lehr

Acknowledgements

I would like to thank my supervisor Prof. Dr. Jesper Larsson Träff for his extensive feedback and help. Although I was able to choose the direction of my work completely freely, I could count on his opinion and advice at all time. In addition to that, he regularly recommended fitting literature and even offered me a place in his office, where I could work undisturbed.

I would also like to show my gratitude to my girlfriend, Katharina Unger, who always listened to my ideas and problems during all times of my thesis. Konrad von Kirchbach, whose master thesis was in a similar field, also encouraged me with his valued input and interesting discussions. Without them motivating me regularly, I would not have been able to finish this work as quickly as I did.

Lastly, I would like to thank my family, even though they are not too familiar with the details of this thesis. Their support - both financially and emotionally - allowed me to solely focus on my studies throughout all of my bachelor and master courses.

Kurzfassung

In dieser Diplomarbeit stellen wir drei Algorithmen vor, mit denen wir das Mapping Problem zwischen Knoten aus kartesischen Graphen und physischen Prozessoren eines Clusters lösen. Prozesse in diesem rechteckigem, kartesischen Gitter kommunizieren miteinander entlang von festgelegten Mustern, sogenannten Stencils. Die Problemstellung ist nun ein Mapping zu finden, bei dem diese Kommunikation möglichst oft zwischen Prozessen passiert, die auf der gleichen Maschine sind und daher über den schnellen Arbeitsspeicher anstatt dem langsameren Netzwerk kommunizieren können. 2018 konnte Gropp mit seinem Ansatz im Vergleich zum Standard-Verhalten vom Message Passing Interface (MPI) zeigen, dass eine gute Mapping-Strategie zu Performance-Gewinnen führen kann. Wir vergleichen unsere Ergebnisse daher nicht nur mit MPIs Standardverhalten, sondern auch mit Gropps Algorithmus. Unser Hauptbeitrag zu diesem Mapping Problem ist jedoch die breitere Anwendbarkeit auf inhomogene Systeme und nicht primär ein Performancegewinn im Vergleich zu Gropp.

Um die Qualität der unterschiedlichen Mappingstrategien zu vergleichen, stellen wir zwei Optimalitätskriterien vor, die die bereits erwähnte Annahme, dass Kommunikation über den RAM schnell ist, verkörpern. Für unsere Messungen verwenden wir die nicht-blockierenden `MPI_Irecv` und `MPI_Isend` Funktionen. Unsere Ergebnisse zeigen, dass alle 3 unserer vorgestellten Algorithmen, im Gegensatz zu Gropp, für alle Situationen Mappings finden konnten. Für Instanzen in denen auch Gropps Algorithmus funktioniert, können wir mit unseren Ansätzen gleich gute oder sogar bessere Laufzeiten feststellen.

Der erste unserer Algorithmen teilt das kartesischen Grid rekursiv inzwei. Diese Graphen Partitionierungsstrategie ist weit verbreitet und ermöglicht uns auch hier, eine logarithmische Laufzeit in der Anzahl der Prozesse zu erzielen. Unsere Implementierung erlaubt es uns außerdem, eine obere Schranke, bezüglich der Qualität der gefundenen Lösungen zu definieren. Mit einer abgewandelten Form dieses Algorithmus, zeigen wir, dass auch andere Stencils beim Mapping bedacht werden können. Hierbei wird das Grid mit einer Priorität auf gewisse, Stencil-abhängige Dimensionen gespaltet.

Der dritte Algorithmus ist aus Beobachtungen hervorgegangen, wo das Standardverhalten von MPI unter gewissen Bedingungen sehr gute Mappings liefert. Hierbei wird das Grid in rechteckige Streifen geschnitten, welche dann mithilfe des Standard Algorithmus befüllt werden. Obwohl die theoretische Oberschranke dieses Algorithmus schlechter ist, als die des Ersten, liefert dieser Ansatz oft kompaktere Mappings.

Abstract

In this thesis we introduce different algorithms for mapping physical processes to Cartesian grids. We assume that processes within this grid communicate with certain neighboring processes as defined by a given stencil. We show that this mapping problem is already NP-complete for two dimensional grids and a very simple isomorphic neighborhood. We compare the current state of solutions in the field of High Performance Computing, specifically the Message Passing Interface (MPI). With his algorithm from 2018, W. D. Gropp showed promising performance, which is why we compare our approaches to his work, as well as MPI's standard behaviour. For qualitatively comparing concrete mappings, we define fitting optimality criteria, based on the core concept of minimizing inter-computation-node communication. We benchmarked using `MPI_Irecv` and `MPI_Isend` and show that our algorithms can find mappings, where Gropp cannot and that we could match or improve upon his resulting mappings in terms of quality and runtime.

The first algorithm, which we present is similar to other graph partitioning approaches, since it utilizes recursive splitting in order to guarantee a logarithmic runtime wrt. the number of vertices in the Cartesian grid. We also guarantee a quality bound for this algorithm, which becomes better with increasing number of dimensions. In another implementation-variant, we adapted this algorithm for accommodating differently shaped stencils by weighting dimensions before the recursive splitting depending on how much communication happens across them.

The third approach attempts to find hyperrectangular strips within the Cartesian grid, which are then filled similar to MPI's default row-major rank assignment. Although the theoretical bound of this approach is not as good as the first, this assignment strategy yielded the most compact mappings most of the time. Its main shortcoming, however, is its inability to adapt these strips, depending on different stencils.

Contents

CHAPTER 1

Introduction

Many High Performance Computing (HPC) problems are structured as n-dimensional Cartesian grids. The performance of the implied, sparse data exchange is problem-specific and depends on multiple factors. On distributed memory systems these factors include not only the scheduling of the messages themselves, but also the placement of virtual processes to physical processors. Often the desired communication follows some structural patterns, such as stencils. This means that processes communicate with other processes along fixed relative offsets within the given n-dimensional grid. Examples for such patterns can be found in the field of Computer Graphics, such as edge detection filters, which have been implemented in MPI by Aithal et al. [1] or in various iterative methods for solving linear systems, such as Jacobi- or Gauss-Seidel stencils [24].

The Message Passing Interface (MPI) [19] offers various functionality to support such communication, as well as the interface to allow for process remapping, however, current implementations do not leverage these possibilities. Gropp [7] presented an overview over some systems and whether they utilize the process remapping features of the `MPI_Cart_create` function. In Table 1.1, it can be seen that none of the surveyed systems currently leverage the remapping feature. The current MPI interface also does not cover the majority of common use-cases but offers either too specific interfaces that have a very narrow applicability or too general methods that cannot convey the structural properties of the given stencil operation.

Another problem that hinders novel approaches from becoming widely-spread is the fact that currently there are next to no benchmarks that would make different implementations comparable. Conversely, this is probably due to the lack of current implementations that leverage these features. This problematic duality shows that it would be necessary and important to offer not only an interface or algorithm to solve said structured communication problems, but also to offer the means to compare these approaches to other solutions.

Table 1.1: None of the surveyed systems remap the processes. MPICH and Open MPI were included, as they are in use on many systems. Taken directly from Gropp [7].

Name	System Type	MPI	Cart Remaps
Blue Waters	Cray XE6/XK7	Cray MPI	No
Theta	Cray XC40 (KNL)	Cray MPI	No
Piz Daint	Cray XC40/XC50	Cray MPI	No
(Open Source)		MPICH 3.2.1	No
(Open Source)		Open MPI 3.1.0	No

1.1 Problem Description

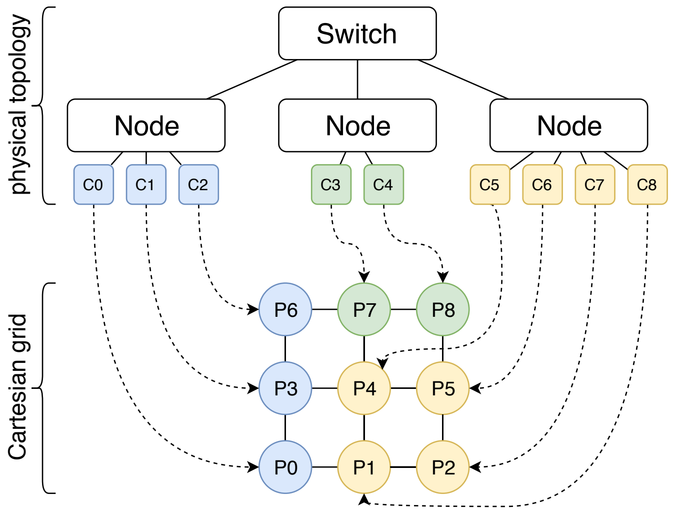


Figure 1.1: A visualization of the problem description. Each computation node's processing cores have to be assigned to the processes of the application specific Cartesian grid. The task lies within finding such an assignment (dotted line), such that the overall communication time is minimized.

The machines and clusters, which are typically used in High Performance Computing have a setup, where different physical cores may have varying connections to other cores. Latency and bandwidth are traditional performance indicators that are accounted for when analysing different connections. A priori, one might conclude that two given communicating cores have a higher bandwidth and a lower latency when they are located in the same processor, compared to two cores that are not located on the same socket or not even on the same machine. Not every HPC cluster is set up with the same network connectivity, so the resulting hardware-topologies may differ between setups. With that in mind, each physical core can be visualized as a node in the topology, where the (weighted) edges between the nodes correspond to the differently performing connections between any two cores.

On the other hand, there are the applications, which run on these clusters. Some applications contain communication patterns that favour different physical topologies more than others, depending on the use case. In this thesis, the focus lies on programs that contain stencil computation, where an application-wide, strictly defined communication pattern is followed by every process. Additionally, we restrict the processes to be arranged in Cartesian grids, meaning that every process is assigned a coordinate within a fixed rectangular grid, not allowing any deviations in shape. We can now visualize each process as a node in this virtual communication graph, where each edge corresponds to an entry to the defined communication pattern.

Now we essentially have two different graph-views of the problem instance, which is visualized in Figure 1.1. One with the physical machine setup and one with the application's communication pattern in mind. The obvious question now is: How can we map the virtual processing cores from the application-communication-graph onto the actual physical-topology, so that the communication time - and therefore the overall runtime - is minimized? Gropp [7] showed with his experimental work that there is a lot of potential in efficiently mapping these two graphs, but before we analyze his findings, we have to define which properties an *efficient* mapping should have.

1.2 Optimality Criterion

In later sections, we prove some properties our algorithms, but in order to be able to do that, we have to define what these algorithms want to achieve. In the real world, there are small variations in communication-speed between any two cores. Constructing a fully connected graph, where every edge corresponds to the link between these cores, would therefore probably be the most accurate representation. However, we will simplify this aspect and only differentiate between cores that can communicate directly over shared memory and cores that have to communicate over the network. In other words, cores that are seated on the same machine are assigned the same partition. Communication within the same partition is viewed as efficient and any communication between two different partitions is deemed inefficient. This assumption is shared by Tan et al. [29], where the authors even state their opinion about finding good process distributions, depending on the network's bandwidth, or Mamidala et al. [15] who show their findings for their InfiniBand setup.

In HPC, the runtime of an application corresponds to the longest runtime of any participating process. The "bottleneck" process, therefore, is responsible for the whole application's runtime. Similarly, we propose one optimality criterion being as follows:

1. For each partition p, calculate the sum of all communications between processes within p and other processes from other partitions.

2. Take the maximum of these numbers, as it represents the "bottleneck"-partition.

Another criterion, which we propose in addition to the bottleneck, is the sum of all communications between different partitions. Similarly to the first criterion, we calculate it as follows:

1. For each partition p, calculate the sum of all communications between processes within p and other processes from other partitions.
2. Sum up these numbers, as they represent all inefficient communication in the network.

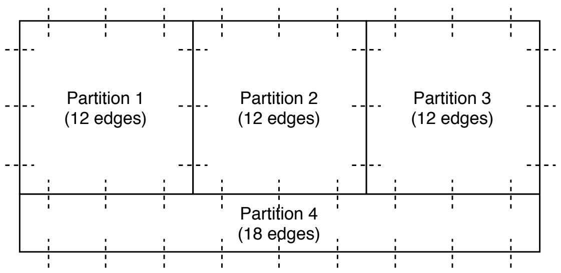


Figure 1.2: A simple example where the edges between these 4 partitions are visualized and counted for evaluation. Note that in this case, communication across both grid-borders is allowed.

For a visualization of these criteria, see Figure 1.2, where 4 partitions with 9 cores each have been mapped to a 4 by 9 grid (Only edges between different partitions are visualized here with the dotted lines.) In this case the application-wide communication scheme only includes communication with direct neighbours along the horizontal and vertical axis (5-point stencil), however, communication across borders is allowed (periodicity across both axes). The quality of this mapping wrt. the first condition would be 18 according to the long, bottom partition. The quality wrt. the second criterion would be 54 as it is the sum of all partitions' outgoing edges $(3 \cdot 12 + 18)$. Both of these criteria are quite similar and solutions with a good "bottleneck" value will not be bad with respect to the other criterion and vice versa. Given the previous assumptions and simplifications from the real world topology, both criteria are expected to be similarly performing indicators.

1.3 Related Work

There are two different aspects, which this thesis focuses on. The first one being the implementation of algorithms for the mapping problem. We will start with more general mapping solutions for MPI, which work for general graphs and proceed to look into more specific approaches. The other aspect is the theoretical approach, including the complexity analysis. This topic is not directly connected to MPI implementations, but rather to mathematics in general.

1.3.1 Mapping Problems in HPC

Hoefler and Snir [8] present heuristics for how to map application communication patterns to the underlying network topology. Their algorithm works for heterogeneous networks, meaning that they allow for differently sized computation nodes. They defined performance metrics, based on worst-case graph congestion and the average dilation, which essentially is the sum of all lengths of paths between two nodes times the frequency that they communicate with each other. They briefly showed the NP-completeness of their problem, assuming however that the input is given in an uncompressed manner. This is completely fine for their approach for general graphs, but their NP-completeness proof is unfortunately not directly applicable to Cartesian grids. They presented three different mapping strategies, one of them being a greedy algorithm. Another, more scalable approach is their recursive bisection algorithm - a strategy, which is commonly used for graph partitioning. They used the SCOTCH package [21] and METIS library [27] for that. Their third approach attempts to shape the adjacency matrices of the communication graph and the network graph as similarly as possible with their algorithm. Their results show that it heavily depends on the underlying system, which of their three approaches yields the smallest congestion.

In their paper "Improving MPI Applications Performance on Multicore Clusters with Rank Reordering", Mercier and Jeannot [17] also work with general graphs and specifically look into the workings of `MPI_Dist_graph_create` - an MPI function, where the user may define edge weights between processes within the given MPI communicator. They distinguish between binding processes to specific cores, which is application independent and has to be done by the user and rank reordering, which is application specific and transparent to the user. Both aspects have their pros and cons, but since libraries should generally aim to be portable, the latter approach is preferred here. They used their own algorithm called TreeMatch [10], which recursively partitions the graph, depending on multiple layers of the system's memory hierarchy. For their benchmarks, the authors tried two different performance metrics - one being the communication frequency and one being the message size - and benchmarked the resulting mappings for both of them.

Both of the previously presented works already show promising results for general graph mapping and there are many others, which focus on different aspects. However, there is little literature in the specific field that this thesis targets, as was earlier emphasised with Table 1.1, where Gropp [7] had shown the scarce (or non-existing) realization of the reorder mechanism for Cartesian grids.

Gropp [7] presented a simple approach for using node information to implement Cartesian topologies. This is very similar to the goal of this thesis, however, his presented factorization method only works for equally sized nodes and he did not present a formal proof for guaranteeing a certain quality of his solution. His excellent experimental results, however, are one of the motivations for this thesis and will be compared against. His algorithm works by performing two-level-decomposition of the grid - a prime factorization for both the size of the computation nodes, as well as the grid's dimensions. The partitions' prime factors are then distributed over the dimensions so that the partitions' dimensions become as square as possible. The workings of his algorithm will be explained in greater detail in a dedicated Chapter 4, where we will also analyze its runtime and quality. An important difference between Gropp's and this thesis' benchmarking is Gropp's inclusion of halo-exchange algorithms. This approach is commonly used in iterative stencil computation for reducing the number of required communication rounds. While this approach certainly is sensible in real-world applications, it is not the primary goal of this work, where the focus lies on purely reducing the communication and communicator construction time.

In a very recent work, Niethammer and Rabenseifner [20] showed a more theoretically sophisticated approach than Gropp with the same core idea and very promising results. In their approach, the authors made the grid dimensions themselves variable, in order to find better mappings. However, we regard the choice of dimensions as part of the user's responsibilities, rather than the algorithm's degree of freedom. Similarly to the already mentioned approach of Mercier and Jeannot [17], the authors' mapping strategy considers multiple layers of communication. Although their algorithm uses a similar multi-level decomposition approach to Gropp, Niethammer and Rabenseifner could sometimes find even bigger improvements, since the grid's dimensions themselves were variable. The impressive experimental results aside, the authors did not provide an upper bound to the quality of their solution.

We distinguish ourselves from these two publications by allowing for differently sized partitions and different stencils. We argue that this improvement is necessary in order for the solutions to replace MPI's current standard behaviour.

Stencil computation itself has also been refined over the years, as parallel, shared memory algorithms were needed to leverage modern GPUs' potential. In the field of HPC, Roth et al. [23] showed optimizations that can be done to the stencil code itself when compiling Fortran applications. However, the focus of this thesis does not lie on efficient stencil computation itself, but on mappings that favour stencil computation in general.

1.3.2 Similar Combinatorial Problems

In the following chapter, we will perform a complexity analysis of our mapping problem, so we compare similar combinatorial problems in this section. At its core, our problem resembles the quadratic assignment problem (QAP), which has been introduced by Lawler [13]. In the QAP, the task is to assign n facilities or factories to n different locations, with a given demand between any pair of facilities and a given weight between any two locations. In our case, the facilities would correspond to the application's virtual Cartesian topology, where the demands are represented by the stencil. The locations and the weights between them correspond to the physical cores and the hardware-links between them. This problem has been shown to be NP-hard by Sahni and Gonzalez [25]. In 1996, Burkard et al. [3] introduced a library called QAPLIB for solving some instances of this assignment problem. Back then, they limited their solution to only work for a maximum of 256 nodes, due to the difficulty of the problem.

Although the QAP is well explored, it is unfortunately not directly connected to our special case. We restrict our “facilities” (processes) to regular, isomorphic Cartesian grids, which intuitively reduces the complexity and space requirements. Additionally, we restrict the “locations” (cores) to either be within the same node or only accessible over the network, which restricts the weights to 0 or 1. A priori, we do not know, whether these restrictions make the problem easier or harder to solve, which is why further analysis is needed.

In an earlier stage of this work, we attempted to solve the problem geometrically and only considered 5pt-stencils. Our assumption - which we will not prove here - was that any optimal mapping, wrt. the sum of outgoing communication edges will only contain connected partitions. With this in mind, we looked at a variety of packing problems. In 2001, Lodi et al. [14] presented an excellent comparison between two-dimensional bin- and strip-packing problems and their connectedness. These problem definitions allow for differently sized rectangles, which is great for our case where computation nodes do not necessarily have to have the same sizes. Unfortunately, the restriction to rectangular shapes is problematic, since we only surely know each partition’s size and not the dimensions of the bounding rectangle. We do not even know, if the optimal mapping consists of rectangular partitions or more irregular shapes, as we will see in later sections.

Since each partition’s shape is variable, we attempted to express our problem as a scheduling problem with malleable tasks. Although there are many variations to the problem statement, generally the difficulty lies within finding the optimal number of machines that concurrently execute any given task and arranging them so that the overall computation time of all tasks is minimized or does not exceed a given deadline. Mounié et al. [18] discuss different approaches for solving this scheduling problem. In one of their approaches, they utilize the knapsack problem, another famous combinatorial problem, which is commonly used for showing algorithmic complexity. Unfortunately, there are a few dissimilarities between the scheduling and our problem again. Even though we can now model the varying sizes of the partitions’ rectangles by changing the number of machines assigned to the corresponding task, we still are limited to rectangular shaped partitions. Additionally, we would have to formulate our optimality criterion into some sort of scheduling constraint, which is usually not possible with a linear expression and therefore quite costly.

A different problem, which allows for non-rectangular shapes is the stock-cutting problem, as explained by Burke et al. [4]. It is often used in the textile and metal-working industry, where irregular shapes have to be cut out of a given stock, ie. a roll of cloth or a sheet of metal, with as little wasted material as possible. In our case, the irregular shapes correspond to the shapes of our partitions, which are now not restricted to rectangles anymore. However, we lost the ability to vary the shapes as part of the problem definition, which means that we would have to know each partition’s optimal shape beforehand.

All in all, there are multiple combinatorial problems, which share some aspects of our mapping problem. However, we could not find any variations, where we could allow for malleable, irregularly shaped partitions, while still being able to somehow encode our optimality criterion.

CHAPTER 2

Definitions

In this chapter, we will cover some technical and theoretical definitions, which will be the basis for future chapters.

2.1 Message Passing Interface (MPI)

The Message Passing Interface (MPI) [19] is a commonly used standard in the field of High Performance Computing. As the name implies, it offers various communication routines so that the user does not have to worry about the connectivity specifics. While this thesis focuses on the algorithmic aspect of the mapping problem, which will be explained in the following sections, it is still important to have a basic understanding of MPI in order to be able to follow the implementation and experimental results.

MPI is used for multi-threaded, distributed memory workloads. An MPI application is comprised of multiple *processes*, which usually correspond to a single hardware thread or processing core. This is not strictly necessary, since cores may be *oversubscribed*, which allows for multiple processes running on the same core, however, for performance reasons this is usually not the case. After the MPI library is initialized, every process accesses its own system memory, meaning that if data from another process is required, it must be transferred via one of MPI's functions.

All communication happens across a *communicator* [19, Section 6.1], which essentially is an ordered collection of processes. Within such a communicator, every process has a unique identifier, called *rank*, which may be extracted from a communicator with `MPI_Comm_rank`. The total number of processes within a communicator may be extracted via `MPI_Comm_size`. All point-to-point communication [19, Chapter 3], like the classical non-blocking `MPI_Send` and `MPI_Recv` or the non-blocking `MPI_Isend` and `MPI_Irecv` [19, Section 3.7], have to be on the same communicator and the sending side has to declare the target's rank, while the recipient has to listen for an incoming message from a specific source rank with a matching receive call.

MPI also offers helpful collective communication procedures [19, Chapter 5]. They can be used to either distribute (e.g. `MPI_Bcast` or `MPI_Scatter`) or collect data (e.g. `MPI_Gather`) from all participating processes. Common map-reduce scenarios such as summation or min/max operations are already implemented and can be used with e.g. `MPI_Reduce`. In this thesis, we will not need most of these operations and the benchmarking will be done using the non-blocking point-to-point functions.

In order to construct a new communicator, `MPI_Comm_split` may be used. As the name suggests, this function can create multiple sub-communicators from the calling one, but it can also be used to create a new communicator that describes the same collection of processes, but with a different ordering. As a starting point, MPI offers a default communicator, which contains all processes, called `MPI_COMM_WORLD`. The presented algorithms in this thesis will, therefore, use `MPI_Comm_split` for altering the ordering of this default communicator in order to increase performance.

Another important feature of MPI is the possibility to describe certain topologies, such as graphs and grids. We will specifically look at `MPI_Cart_create`, which enables the user to describe Cartesian grids with specific dimensions. The `reorder` flag can be used to allow the MPI implementation to undertake optimizations during this mapping phase. In the introductory Chapter 1 we will further look at the poor reception of this flag.

2.2 Proofs

In the following chapters, we will analyze the algorithmic complexity of multiple problems. Specifically, we want to find out whether our problems may be solved in polynomial time or not. See Figure 2.1, which was directly taken from a survey by Laudis et al. [12], for a visual overview of the complexity classes that we will focus on. This chapter will explain the methodology used for these proofs and how we can show their correctness.

2.2.1 Decision Problems

Problems deal with different domains. For example, the problem of finding a clique in a graph may be described with an instance, containing a graph structure and a result, containing a collection of vertices. Another problem, 3-SAT, has an instance containing a logical formula and a truth value assignment for each variable as output. Therefore, comparing these two problems directly is not sensible or possible[1]. Our reductions only work for so-called decision problems.

[1]In the literature, it can often be observed that some authors claim that a specific optimization problem is NP-complete. As we know, an optimization problem cannot be in the class of NP, since we cannot define it as such. However, this phrasing is often used to colloquially say that a computationally equivalent decision problem is NP-complete. Sometimes it can even be seen that full NP-completeness proofs are done using the optimization variant, which is not logically sound.

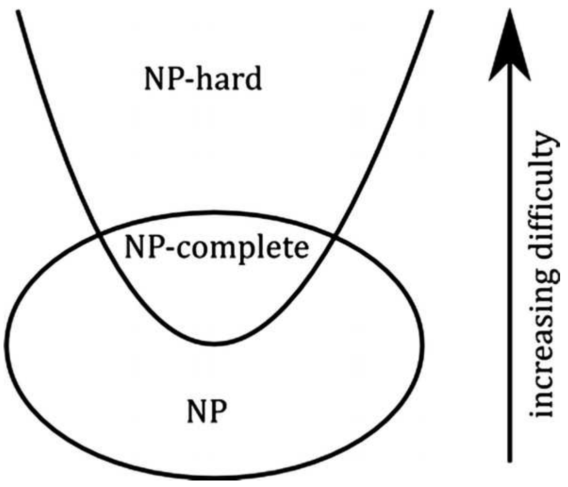


Figure 2.1: A simple diagram showing how two commonly mentioned complexity classes (P and NP) compare to each other. Taken directly from Laudis et al. [12].

Decision problems still have problem-specific instances (graphs or formulae), but they only check for one condition and have a *true* or *false* answer. For example, we can now create a *decision* problem, based on our previous largest clique *optimization* problem, by changing the task from "Find me the largest clique of graph $\mathcal{G}$." to "Is there a clique of size at least k in the given graph $\mathcal{G}$?". To make this rephrasing easier, we introduced another input parameter k (which is common, but not strictly necessary). We can take for granted that every decision problem also delivers a so-called *witness* when the return value is true. If we ask in our example, if a clique of at least size 10 exists and the answer is yes, then we would also be able to get such a clique. We do not know if the returned clique is the biggest or the only one, we just know that it satisfies the condition.

Another necessary step is to show that the newly introduced decision problem is not unrelated to the optimization problem. This can be done by showing that one can be constructed by using the other in polynomially many steps, wrt. the input size of the optimization problem. Using the clique problem as an example again, we can construct the decision problem by calling the optimization problem for finding the largest clique C and then just check, whether $|C| \geq k$ holds or not. Conversely, we can construct the optimization problem, since we know that the largest clique has to be of size $[0 \ldots |\mathcal{G}|]$. We can then call the decision problem for each possible size (or even in logarithmically many steps, if we iterate $[0 \ldots |\mathcal{G}|]$ in a binary search). After constructing the decision problem, we can then proceed by proving its complexity.

2.2.2 Proving Complexity

We chose to perform our proofs by using Turing reductions. More concretely, we will be using Many-one reductions, which are special cases of Turing reductions, presented by Post [22] in 1944.

The idea is similar to any other reduction. Let us suppose we know the complexity of a well-known decision problem $\mathcal{A}$ and want to show that our problem $\mathcal{B}$ has the same complexity. We can reduce problem $\mathcal{A}$ to $\mathcal{B}$ - written $\mathcal{A} \leq_m \mathcal{B}$ - by defining a procedure that would solve $\mathcal{A}$ by using $\mathcal{B}$ e.g. as a subroutine. We can then argue that $\mathcal{B}$ cannot be easier to solve than $\mathcal{A}$, since if it would be, then we would have found an easier way of solving $\mathcal{A}$ using the reduction, contradicting our assumption of $\mathcal{A}$ being in a known complexity class.

When proving NP-hardness, we have to be mindful of the runtime of the reduction process as well. For example, if $\mathcal{A}$ were an NP-hard problem and we want to show the same for $\mathcal{B}$, then a reduction that calls $\mathcal{B}$ exponentially many times does not prove anything. The reason is, that it may very well be that problem $\mathcal{A}$ can be solved using exponentially many steps that individually have polynomial runtime (3-SAT for example). We therefore restrict ourselves to polynomial reductions - written as $\mathcal{A} \leq_p \mathcal{B}$ - which means that within the reduction, problem $\mathcal{B}$ may only be used polynomially many times wrt. the input of problem $\mathcal{A}$. These reductions, which use $\mathcal{B}$ as a subroutine polynomially many times, are also known as Karp reductions.

As already mentioned, we will not use these Karp reductions, but rather Many-one reductions. The difference is that instead of writing a hypothetical algorithm for $\mathcal{A}$ using $\mathcal{B}$ as a subroutine, we will only transform the instances of problem $\mathcal{A}$ to an instance of problem $\mathcal{B}$. As a consequence, we can show that instances of $\mathcal{A}$ may be solved by an algorithm for $\mathcal{B}$ in just one call (which, of course, is still polynomial). This instance transformation has to adhere to two rules. Firstly, the new, transformed instance has to be polynomially bounded by the size of the old instance. Otherwise, we could cheat again and move some of the algorithmic complexity to the instance creation phase itself. Secondly, for NP-completeness proofs, it has to be possible to verify whether a given solution to a specific instance is indeed a valid solution or not. This restriction might have an impact on the instance transformation as well.

After defining such a reduction, we have to verify that the instance transformation $\mathcal{I}^{\mathcal{A}} \rightarrow_p \mathcal{I}^{\mathcal{B}}$ is correct. Therefore, we have to prove that $\mathcal{B}$ returns true on a given instance $\mathcal{I}^{\mathcal{B}}$ if and only if $\mathcal{A}$ returns true on $\mathcal{I}^{\mathcal{A}}$. The best and easiest way to do that depends on which problems were used for the reductions. This may be a direct proof for showing $\mathcal{A}(\mathcal{I}^{\mathcal{A}}) \equiv \mathcal{B}(\mathcal{I}^{\mathcal{B}})$ or by showing the implications individually. There are multiple, logically equivalent ways of doing so, for example $(\mathcal{A}(\mathcal{I}^{\mathcal{A}}) \Rightarrow \mathcal{B}(\mathcal{I}^{\mathcal{B}})) \wedge (\neg\mathcal{A}(\mathcal{I}^{\mathcal{A}}) \Rightarrow \neg\mathcal{B}(\mathcal{I}^{\mathcal{B}}))$ or $(\mathcal{A}(\mathcal{I}^{\mathcal{A}}) \Rightarrow \mathcal{B}(\mathcal{I}^{\mathcal{B}})) \wedge (\mathcal{B}(\mathcal{I}^{\mathcal{B}}) \Rightarrow \mathcal{A}(\mathcal{I}^{\mathcal{A}}))$.

2.3 Cartesian Graphs with Isomorphic Neighbourhoods

A graph $G = (V, E)$ is comprised of a set of vertices, V, and a set of edges between those vertices, $E \subseteq V^2$. In this thesis, we will restrict ourselves to the simpler Cartesian graphs or grids. Cartesian grids only have vertices within a n dimensional Hyperrectangle. Given any list of dimension sizes $D = \{d_1, d_2, \ldots, d_n\}$, where $d_i \in \mathbb{N}^+$ denotes the size of the ith dimension, we can construct V unambiguously using the Cartesian product

$$V = \prod_{i=1}^{n} \{0, \ldots, d_i - 1\}.$$

In grid- or lattice graphs, each vertex v is connected to a set of other vertices, U, via a regular tiling. In future, we will call these connected vertices U neighbours of v, hence the name "isomorphic neighbourhood". Similar to these grids, we define our isomorphic Cartesian graph neighbourhood with a bijection relation between the neighbourhood of each vertex $v \in V$ to the same graph H [9, 28, 30]. We will not explicitly define this relation; instead, we will be using stencils for that.

2.3.1 Stencils

In a stencil computation, matrix elements are updated according to a pattern, which is fixed for every element. This pattern is called *stencil*. Usually, each step in this computation is comprised of simple, arithmetic operations, such as addition or multiplication, however, in this thesis, we focus on the pattern itself, not the actual computation. Figure 2.2 shows 6 different simple, exemplary stencils. Later on, we will describe stencils as a list of vectors for mathematical discussions or as a flat integer array in code. The 5-point (5pt for short) stencil, visualized in Figure 2.2, for example, can be written as $S = [(1, 0), (-1, 0), (0, 1), (0, -1)]$ or in a flat integer array as $S = [1, 0, -1, 0, 0, 1, 0, -1]$.

In Figure 2.2 we visualized all 6 stencils with directed edges, originating from the center node. This directionality can be read as "Node x requires data from node y". This behaviour can be symmetric, as is the case with the 5pt-stencil for example. Each node requires data from, e.g., its right neighbour, but its right neighbour also requires data from the original node, as it is the right node's left neighbour. The asymmetric stencil shows a counterexample to that, where the top-right neighbour does not require data from the center one since the stencil contains no outgoing edge to the bottom-left node. This notation of symmetry of directionality is different to how we view it in MPI, where we have to consider the whole data-flow. If we look at the asymmetric stencil again, the top right neighbour has to know that it will need to send data to its bottom left neighbour, although this connection is not directly part of the stencil. We could therefore argue, that modelling asymmetric stencils is not sensible, but we will keep them in for completeness sake.

Stencils are commonly comprised of weighted vectors, indicating their impact on the computational kernel. The horizontal edge detection filter from Figure 2.2, for example, usually weights the two horizontal edges with 2 and the diagonal edges with 1, indicating that the horizontal neighbours' values are twice as important for the computation as the others. There is an important distinction between this *weight* and the same term when it is used in conjunction with MPI or graph mapping in general. Niethammer and Rabenseifner [20], for example, proposed an algorithm where the edges' weights impact the mapping by prioritizing a clustering of nodes, which are connected with higher edge weights. These weights, however, focus on the different *amounts* of data, which have

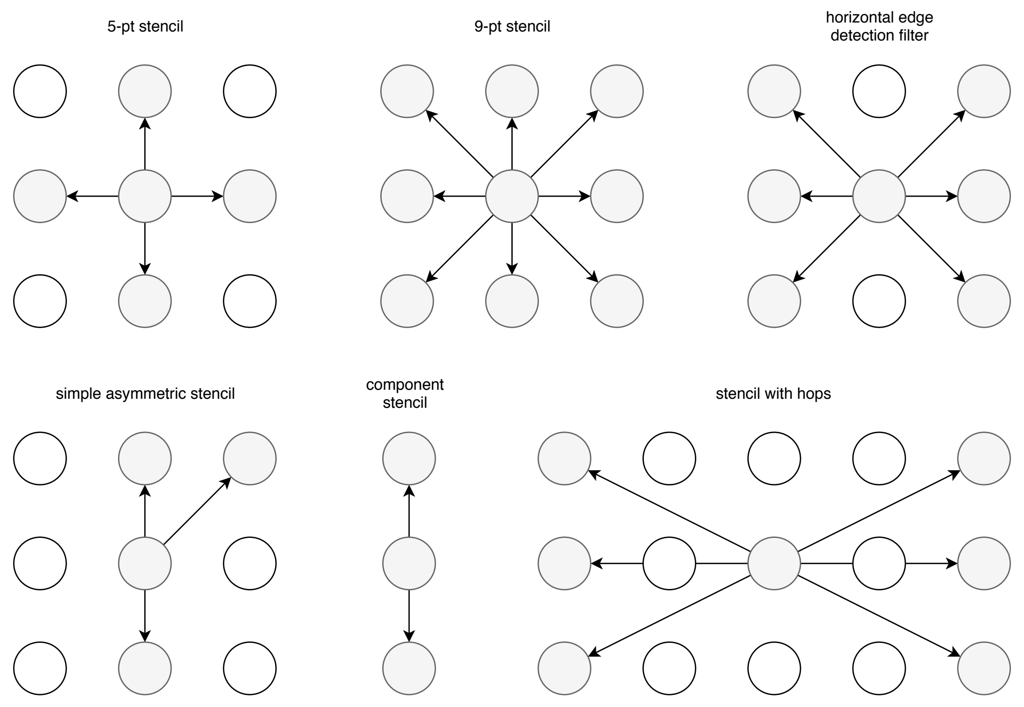


Figure 2.2: 6 different stencils. The 5- and 9-point stencils are most commonly referred to and are often used for averaging out elements in matrices. The horizontal edge detection filter is used in computer graphics as a part of the Sobel filter[26]. The other three stencils show that patterns do not have to be symmetric and may even split the graph into multiple components, as is the case with the presented component stencil.

to be transmitted, not on the *importance* of the values themselves. In the case of the horizontal edge detection filter, for example, Niethammer and Rabenseifner's weighting approach would not make sense, since while the horizontal edges transmit more important values, the transmitted amount is the same for all neighbours.

2.3.2 From Stencils to Cartesian Graphs

Given a set of nodes V and a stencil S, containing $|S|$ n-dimensional vectors $s_1, \dots, s_{|S|}$, we can now construct the edges of our graph G as follows:

$$E = \{(v, v+s) | v \in V, s \in S, v+s \in V\}$$

Note that this construction does not treat edge nodes differently from nodes within the grid. The last condition, $v + s \in V$, simply ensures that all edges in E are between actual nodes in V. Any given problem usually not only demands a specific stencil but also dictates the behaviour along the edges. Sometimes a default value, like 1 or 0, is assumed or the edges two vertices from opposite sides of the grid, forming a torus shape.

This special treatment also does not have to be equal for all edges. For example, in a 2D grid we could have edges across borders on the first dimension, but not on the second.

At the beginning of this section, we already presented a method for constructing the graph's vertex set V from a list of n dimensions. We still have to describe the bijective relation between each vertex's neighbourhood and H, where H is a helper graph, induced by the stencil. Given a stencil S, containing $|S|$ n-dimensional vectors $s_1, \dots, s_{|S|}$, we can formally construct this graph H with

$$H = (V', E'),\ V' = \{s | s \in S\} \cup \{0\}^n,\ E' = \{0\}^n \times \{s | s \in S\}.$$

Disregarding the distinction of different edge cases, we can now construct the bijective relation for all vertices of our graph $G = (V, E)$ and the helper graph $H = (V', E')$. This bijection directly follows from the previous construction of our edge set E.

$$\forall v \in V,\ \forall (v, u) \in E : \quad v \mapsto u - v$$

A visualization of this process can be seen in Figure 2.3. In this example, the simple 2D component stencil was used to induce the visually similar helper graph H and the bijection relation between a selection of nodes of G and said helper graph is visualized using the curvy edges.

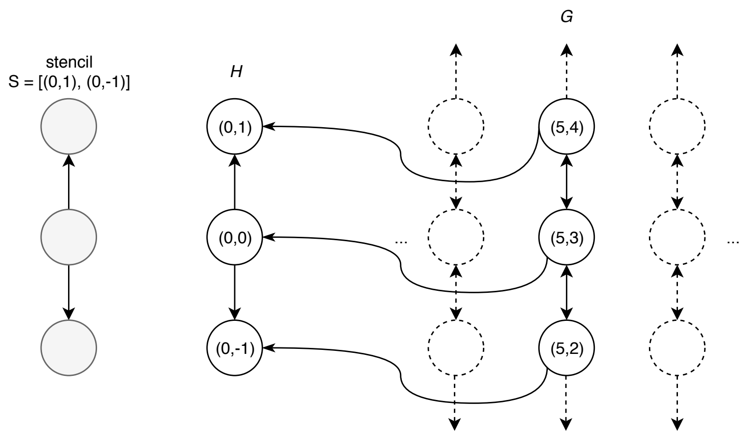


Figure 2.3: The interaction between a simple stencil, the induced helper graph H and its isomorphic properties are shown here for a portion of the graph G.

In this section we not only explained what Cartesian grids and graphs are but also how we can construct them, given a list of dimensions D and a stencil S. This is important for later chapters since we will only describe the graphs implicitly with these variables, instead of enumerating the vertex and edge sets explicitly. While this notation makes the graph representation more compact, this also means that proofs regarding algorithmic complexity will become more complicated. This is because the number of vertices within the graph is not polynomially bounded by the *number* of dimensions but by the *sizes* of each dimension. These complexity pitfalls will be further analyzed in Chapter 3

CHAPTER 3

Complexity Analysis

In general, graph partitioning or graph embedding problems are NP-hard, as shown by Bokhari [2]. However, in the context of this thesis, the given graphs and partitions adhere to many restrictions, which could make some aspects of the problem easier to solve. Not only does *easier* mean that some NP-hard problems may become polynomial in complexity, but also that - in the case of NP-hard sub-problems - approximation algorithms may either become more accurate or faster. In this chapter, the focus lies on defining different sub-problems and restricting their properties, followed by the respective complexity analysis.

These restrictions should not be made arbitrarily, so for each sub-problem, the scope of possible patterns, partitions and dimensions, as well as the optimality criterion have to be well defined. Note that, for example, to show that a problem is NP-hard for any dimension, it suffices to show that it is NP-hard for two dimensions. Additionally, the problem at hand is an optimization problem, with the goal of finding the best possible mapping of processes to partitions. It therefore cannot be NP-complete, since NP-membership is only defined for decision problems and has to be used carefully in reductions. For this sake, a practically equivalent decision problem will be defined.

3.1 Formal Problem Definition

The formal definition of the partitioning problem allows for well structured proofs. The optimization variant, is defined as follows:

GRID-PARTITIONING

INSTANCE: A stencil S, which is a list containing n-dimensional vectors that serve as the communication pattern between processes.
The dimensions D, which is an array of length n, consisting of the sizes of the n dimensions of the grid.
A list of partition sizes P that contains all positive integer sizes for the partitions that have to be filled. P is st. $\sum p \in P = \Pi d \in D$
A list of boolean variables T of length n, consisting of the periodic behaviour along dimensions. $t \in T$ is $false$ (or 0) iff no communication is accounted for around the grid border, $true$ (or 1) otherwise.

QUESTION: Find the optimal mapping of processes in the grid to partitions, subject to *a)* the bottleneck partition, *b)* the sum of off-node communications.

An example instance could be **GRID-PARTITIONING**$(S = [(-1,0),(1,0)], D = [5,4], P = [10,10], T = [0,0])$. In words this instance means that we are working with a 5 by 4, two-dimensional grid (defined by D) which has to be mapped to two partitions, both having size 10. An optimal mapping has then to be found with respect to the given stencil S, which in this case only defines communication along the first dimension and only to the direct neighbours. Colloquially speaking, every process communicates with its left and right neighbour.

For this instance, a possible mapping of processes $c_{x,y}$ to partitions p_i could be:
$p_0 = \{c_{x,y} | x \in \{0..4\}, y \in \{0..1\}\},\ p_1 = \{c_{x,y} | x \in \{0..4\}, y \in \{2..3\}\}$,
which colloquially speaking translates to assigning the bottom two rows to one partition and the top two rows to the other. A visualization of this instance and mapping is shown in Figure 3.1.

In order for reductions to be formally correct, let the corresponding decision problem be formulated as follows. Note that only the parameter Q was added to the instance, in order to describe the target quality of the solution, with respect to the defined optimality criterion.

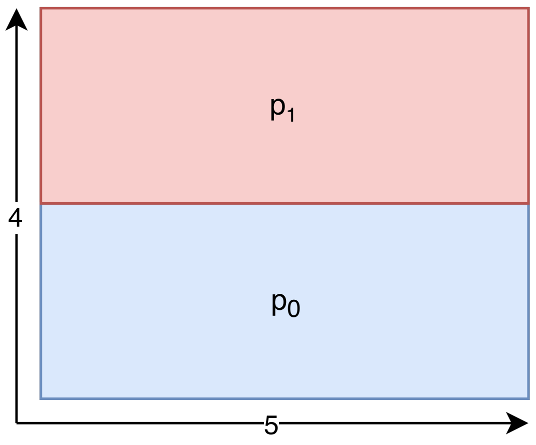


Figure 3.1: A feasible solution to the partitioning instance **GRID-PARTITIONING**$(S = [(-1,0),(1,0)], D = [5,4], P = [10,10], T = [0,0])$. This mapping's costs, wrt. the two optimality criteria is 5 for the bottleneck and 10 for the sum of all communication edges.

BETTER-GRID-PARTITIONING

INSTANCE: A stencil S, which is a list containing n-dimensional vectors that serve as the communication pattern between processes.
The dimensions D, which is an array of length n, consisting of the sizes of the n dimensions of the grid.
A list of partition sizes P that contains all positive integer sizes for the partitions that have to be filled. P is st. $\sum_{p \in P} p = \prod_{d \in D} d$
A list of boolean variables T of length n, consisting of the periodic behaviour along dimensions. $t \in T$ is $false$ (or 0) iff no communication is accounted for around the grid border, $true$ (or 1) otherwise.
An integer Q that stands for the target quality of the solution (wlog. less is better).

QUESTION: Does there exist a mapping of processes in the grid to partitions, whose quality (with regard to the defined optimality criterion a or b) is less than or equal to Q?

To show that this decision problem is practically equivalent to the optimization problem, we define both using the respective other one. The optimization problem has some known, trivial bounds for the solution quality Q, depending on the optimality criterion. For example, Q could be between 0 and the size of the biggest partition times the stencil size. Then the **BETTER-GRID-PARTITIONING** problem can be called in a binary search fashion until the best "yes" instance is found. Conversely, the decision problem

Table 3.1: An overview of the complexity classes of different variations of the mapping problem. We show that for two dimensions and a simple component stencil, the problem becomes NP-hard already.

Partition Sizes	Dimensions	Stencil	Periodicity	NP-hard	NP membership
any	2	component	any	X	X
equal	2	component	any		X
equal	2	5pt	any	?	?
any	any	any	any	X	?

can be formulated using the optimization problem, by simply calling it and comparing the resulting quality Q_{best} to the given parameter Q and returning "yes" or "no" accordingly.

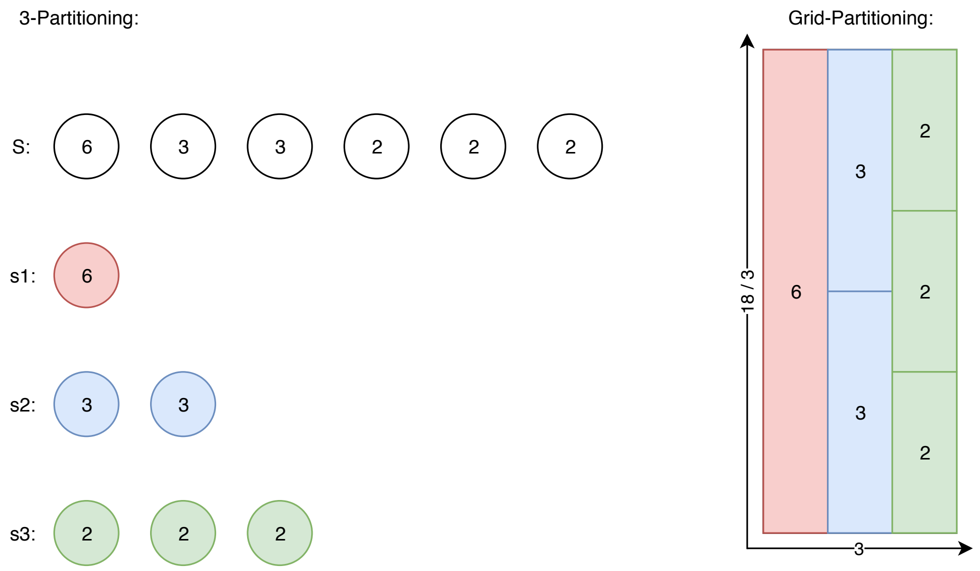


Figure 3.2: A positive 3-way-Partitioning instance $S = \{6, 3, 3, 2, 2, 2\}$ is visualized as an optimal Grid-Partitioning. Each integer i corresponds to one partition p, where the size of p is equal to i. The cost wrt. the sum of all edges is 6.

In the following sections we will use these definitions for proving the complexity of different sub-problems. Table 3.1 presents an overview of the different complexities. Most importantly, the first line states that the mapping problem is already NP-hard for two dimensions and a simple, one-directional component stencil, which we already presented in Section 2.3.1. The proof for this can be seen in Section 3.2. From this, we can derive that the mapping problem has to be NP-hard in general. Additionally we show in Section 3.3 that further simplifications allow us to solve the problem in polynomial time, if the partition sizes are restricted to having equal sizes.

3.2 Any Partition, Two Dimensions, Up-Down Pattern

In this section, we will show that for two dimensions and a very simple pattern, the problem is already NP-complete. Additionally, it can be shown that the complexity does not change, regardless of periodicity, with some adaptations to the proof, which can be seen in Section 3.2.1. This is shown via a reduction from 3-way-Partition, an NP-hard problem, which has been presented by Korf [11], to our **BETTER-GRID-PARTITIONING** decision-problem[1]. To start with, let us define an optimal solution as the one, where the sum of all connections leaving any partition has to be minimized. A similar proof can be found for minimizing the maximum number of outgoing connections per partition - the bottleneck value so to say. See Figure 3.2 for a visualized example of mapping a positive 3-way-partition instance to a grid-partitioning instance, which makes following the proof easier.

Recall that 3-way-Partition is the task of dividing a multi-set of integers into three subsets, that have to have an equal sum of elements. More formally, the problem can be formulated as follows:

3-WAY-PARTITION

INSTANCE: A list, or (multi-)set S of integers. Note that for simplicity's sake, the trivial negative case where $\sum s \in S$ is not divisible by 3 is immediately discarded, since that makes a 3-way-partitioning impossible.

QUESTION: Can S be partitioned in 3 sub-sets $s_{1,2,3}$ where $\sum x_1 \in s_1 = \sum x_2 \in s_2 = \sum x_3 \in s_3$ holds.

Theorem 3.2.1. *The **GRID-PARTITIONING** problem is NP-hard, when restricted to two dimensions, a one-dimensional component stencil* $[(-1,0),(1,0)]$ *and no periodicity.*

Proof. Given an arbitrary instance S' of **3-WAY-PARTITION**(S') we construct an instance of **BETTER-GRID-PARTITIONING**(S,D,P,T,Q) as follows:
$S = [(0,1),(0,-1)]$ (In words, S consists only of the two-dimensional *up* and *down* vectors.),
$D = [3, \frac{\sum_{x \in S'} x}{3}]$ (In words, there are two dimensions, one being 3 wide. The rectangle constructed by these two dimensions has the area required for fitting all partitions exactly.),
$P = \{p | p \in S'\}$ (In words, every element s in the multi-set S correlates to the size p of a partition in P.),
$T = [0,0]$ (In words, there is no periodicity),
$Q = 2 \cdot |S| - 6$

[1]The 3- or multi-way partitioning problem is often confused with the well-known, strongly NP-complete problem 3-Partition, which has been classified by Garey and Johnson [5]. In this problem, a multiset containing $3m$ integers has to be divided into m triplets, which have to have the same sum.

A *yes* instance of **3-WAY-PARTITION** is now corresponding to a *yes* instance of **BETTER-GRID-PARTITIONING**. The simple *up* and *down* stencil makes it so that an optimal partition has the shape of a strip, only stretching across the second dimension and being 1 wide in the first dimension. Such a partition has exactly 2 outgoing connections, one at the bottom and one at the top of the partition. Since there is no periodicity, the top and bottom edges of the rectangular grid, result in every optimal partition touching them having only 1 outgoing connection instead.

We know for a fact that it is possible to find such an optimal mapping because a positive 3-way-partition instance allows us to construct such a mapping by assigning every node across the first index in the first dimension to the partitions that correlate to the values in the 3-way-partition subset s_1. Nodes along the second and third index are put into the partitions that correlate to the values in s_2 and s_3 respectively.

Furthermore, we know that such a mapping always has a quality of $2 \cdot |P| - 6$, since only optimally shaped partitions occur in the best solution, for which we showed the number of outgoing connections earlier. Since this value is equal to Q, which we passed as a parameter to **BETTER-GRID-PARTITIONING**, we successfully constructed a positive instance.

With a similar argument, the other direction can be shown, where a *yes* instance of **BETTER-GRID-PARTITIONING** corresponds to a *yes* instance of **3-WAY-PARTITION**. If there exists a mapping with a quality less than or equal to $2 \cdot |S| - 6$, then that means that every partition is optimally shaped into a vertical strip that only spans the second dimension. Since such a one-wide strip cannot span multiple indices in the first dimension, we know for sure that it can be unambiguously assigned to the corresponding subset s_i, where i denotes the index of the first dimension that the partition is in. Since the collective height of all partitions for one index within the first dimension is exactly equal to a third of the sum over all integers in S', we know that all subsets $s_{1,2,3}$ are equal in size, by construction. □

From this proof, we can formally conclude that not only this restricted problem is NP-hard, but that the Grid-Partitioning problem, in general, is NP-hard as well. Nevertheless, it may be interesting to explore other harder or even easier sub-problems.

3.2.1 Added Periodicity

The proof in the previous section was done without allowing communication to wrap around the edges of the constructed rectangle. Using the same reduction as before, it can be easily seen that adding periodic communication to the first dimension does not change the problem statement since the pattern only foresees communication along the second dimension. In this case, T would simply be changed to $[1, 0]$ and the rest of the proof remains unchanged.

For the second dimension, however, the proof has to be adapted, when periodicity is added ($T = [\_, 1]$). For the same optimality criterion (the sum of all outgoing connections), the change is that Q is no longer set to $2 \cdot |S| - 6$, but $2 \cdot |S|$. Additionally, the allowed integers in the 3-way-partition multi-set have to be restricted to being strictly smaller than a third of all the values; or expressed more formally: $\forall x \in S: \; x < \frac{\sum S}{3}$.

This restriction only removes some trivial cases, since if a single value x would be bigger than a third of the total sum, there would not be a possible 3-way-partitioning anyway, since the partition containing x is already bigger than the other two partitions. Would x be equal to a third, then this value would have to be in its own partition, since there would be no room for other integers. The rest of the problem could be solved with an analog reduction from the partition problem, where only two, equal-sized partitions have to be found, using the multiset $S \backslash x$.

Still, this restriction is necessary, because when looking at the example in Figure 3.2, we can see that the formula $2 \cdot |S|$ does not hold. The partition with size 6 does not have any vertical neighbours and therefore 0 outgoing edges. Other than that, the rest of the proof becomes easier instead, since every optimally shaped partition now has exactly 2 outgoing communications, regardless of the top and bottom edges.

Furthermore, the proof has been done with the sum of all outgoing communications as the optimality criterion. However, with the added information about periodicity, we can express a similar proof, where the number of outgoing connections in the worst partition is minimized. The reduction is analog to how it was done in Section 3.2, but $T = [0, 1]$ and $Q = 2$. Since we know that an optimally shaped partition always has exactly two outgoing edges and more than that otherwise, the correctness of the reduction can be shown trivially. The NP-hardness of the problem without using the periodicity property is also possible using a similar argument, but not as straight forward, since more case distinctions need to be made.

Corollary 3.2.1. *Regardless of periodicity, the **GRID-PARTITIONING** problem is NP-hard, when restricted to two dimensions and a one-dimensional component stencil* $[(-1, 0), (1, 0)]$.

3.2.2 NP-Completeness Proof

There are multiple ways for showing NP-membership of a problem. Traditionally one has to argue that a possible witness solution for a given problem is polynomially balanced in the size of the input and that it can be verified in polynomial time. Since **BETTER-GRID-PARTITIONING** only describes the grid implicitly with its dimensions, such a witness may blow up exponentially, since every node is assigned a partition, not every dimension. For example, a problem instance with three dimensions, each of size 10, implies that the grid has 10^3 nodes, which then have to be assigned to partitions. Describing such an assignment explicitly - meaning to enumerate every mapping in the solution - therefore would not result in a polynomially balanced representation of a witness. This does not automatically mean that our problem is not in NP, it just means that we have to find another way.

Instead of the trivial enumeration, it may be possible to utilize some other properties of the problem. Let us recall the optimization criterion of the problem: "Does there exist a mapping of processes in the grid to partitions, whose quality (with regard to the defined optimality criterion) is less than or equal to Q?". We asked this question where optimality meant minimizing the sum of all outgoing connections per partition and the periodicity was not important. Let us assume (for simplicity's sake) that we want vertical periodicity by setting T to $[0, 1]$. We now know that the optimal solution to this mapping problem has at best $Q = 2 \cdot |S|$, as already explained in the two previous sections. We also know that optimally shaped partitions are one-wide, long strips. If that is not possible, these strips will wrap around the edges and continue in the next index of the same dimension. This increases the number of outgoing connections by two, since the *strip* is no longer 1 wide, but 2. Should the strip still not fit and beginning a new column is needed, no further outgoing connections are added since the middle column is solely occupied by the same partition and does not introduce any new communication. Figure 3.3 represents these different mappings.

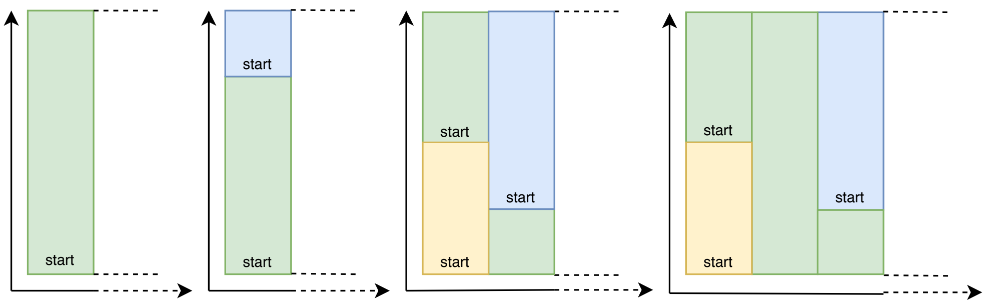


Figure 3.3: Possible instances for the simplified sub-problem, where only vertical communication is allowed across a 2-dimensional grid. Even though the partition sizes may be different, the different possible mappings are still manageable.

Looking at these different mappings, it can be seen that a given partition can be sufficiently described only with the first node together with the partition's size. This representation now is a lot more compact and, in fact, polynomially bounded in the size of the input - concretely in the sizes of partitions P. An example of this representation can be seen for the problem instance **BETTER-GRID-PARTITIONING**$(S = [(0, -1), (0, 1)], D = [3, 5], P = [2, 3, 4, 6], T = [0, 1], Q = x)$ in Figure 3.4. The compact representation of this possible solution can be written as $Sol = [(1, 2), (0, 4), (0, 0), (1, 4)]$, where the tuple at index i corresponds to the starting point of the partition, whose size is described at $P[i]$.

We now also have to check, whether this polynomially bounded representation can be verified in polynomial time. Colloquially speaking, we have to calculate, whether this given witness results in "Yes" or "No" for the decision problem in polynomial time. This can be done by calculating the number of edges that will be created when the ith partition, with size $P[i]$, starts at $Sol[i]$. The details of this calculation will be omitted since it just contains a sequence of simple divisions and modulo operations.

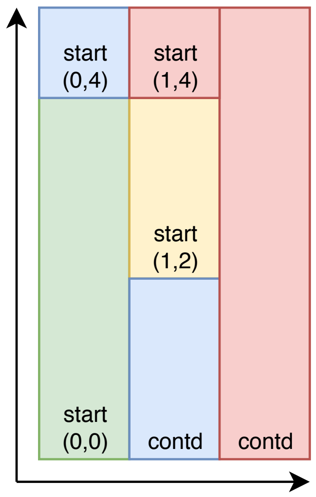


Figure 3.4: A possible solution to the problem instance **GRID-PARTITIONING**$(S = [(0,-1),(0,1)], D = [3,5], P = [2,3,4,6], T = [0,1])$ together with the starting points of the compact representation.

We have now shown that this restricted version of our decision problem is NP-complete. However, this does not mean that the general problem (with unrestricted stencils) is NP-complete as well. In fact, since such a compact solution cannot be found easily, would suggest that this problem is outside of NP.

3.3 Equal Partitions, Two Dimensions, Up-Down Pattern

In the previous section, the NP-hardness of the sub-problem that allows any partition size was shown. In this section, the problem is further simplified, by allowing only partitions with equal sizes of p. With the previous reduction in mind, it might not come as a surprise that this problem is in fact solvable in polynomial time. For simplicity, let us look at the optimality criterion where the number of outgoing connections of the worst partition (bottleneck) is minimized when vertical periodicity is allowed. Similar arguments can be made when periodic communication is not permitted or other optimization criteria are chosen, but more case distinctions are necessary in this case.

Let us make the following case distinction, which is also visualized in Figure 3.5:
Case 1: The second dimension d_2 is a multiple of p; or more formally: $d_2 \text{modp} \equiv 0$.
Case 2: The second dimension d_2 is not a multiple of p and $p < d_2$.
Case 3: The second dimension d_2 is not a multiple of p and $p > d_2$.

The first case is trivially solvable in polynomial time, regardless of the optimality criterion, since every partition can be optimally shaped into a long strip again and greedily packed along the second dimension. Every partition then has two outgoing communications, with small exceptions depending on the periodicity across the top and bottom edge of

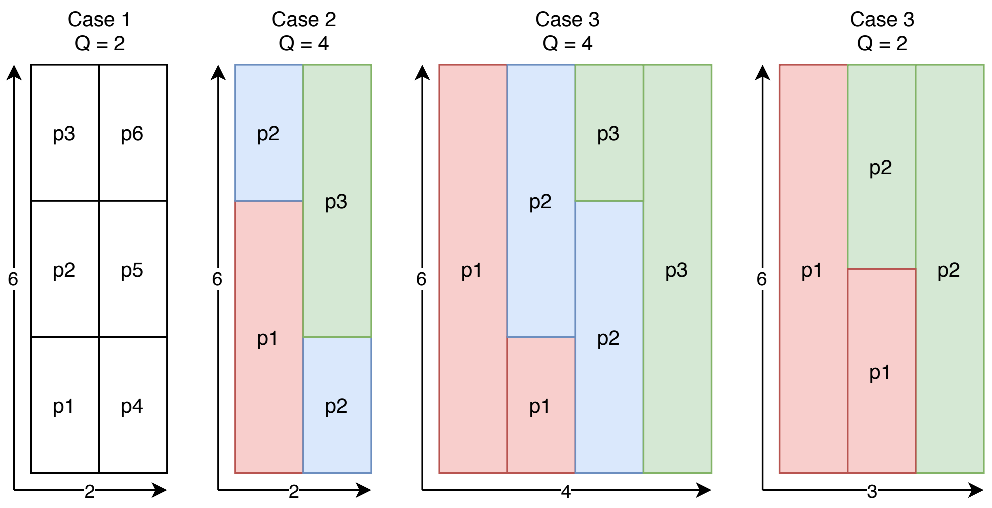


Figure 3.5: Possible instances for the simplified sub-problem, where only vertical communication is allowed across a 2-dimensional grid. The quality Q of each instance is calculated as the bottleneck value of outgoing connections, according to the case distinction.

the rectangle. The number of outgoing connections of the worst partition (bottleneck Q) is 2, in this case.

The second case is not so trivial since it is not possible to form only optimally shaped partitions. Q therefore has to be strictly greater than 2. It must have been impossible for some partition x to be optimally shaped and filled into a strip along the second dimension. However, by filling the partition with all remaining processes from one index in the first dimension (along a strip) and filling the partition with as many processes as needed from the next index in the first dimension, we can minimize the damage. Since $p < d_2$, we know that partition x spans exactly two different indices across the first dimension, leaving us with a bottleneck value Q of 4.

The third case is similar to the second one, with the difference that every partition spans at least two indices across the first dimension. Since all processes along any index in the first dimension are distributed across at most two partitions and all partitions have at most two indices that are not solely occupied by themselves, the bottleneck value Q is bound by $2 \cdot 2 = 4$ as well. If it is possible, however, that all partitions only occupy one index that is shared with another partition, then Q is equal to only 2. This case happens if $(2 \cdot p)\%d_2 = 0$ is satisfied.

With this algorithm, we have shown that a previously NP-hard problem becomes easy to solve by changing just one property of the problem definition. This emphasizes the need to be exact when defining the given problem.

Lemma 3.4. *The **GRID-PARTITIONING** problem can be solved in polynomial time wrt. the input, when restricted to two dimensions, a one-dimensional component stencil*

$[(-1,0),(1,0)]$ *and equal-sized partitions.*

3.5 Equal Partitions, Two Dimensions, 5-Point Stencil

This section deals with the first practically relevant case, which is a mixture between the ones, which were analyzed in the previous sections. We keep the partitions sizes restricted, as we did in Section 3.3 for making the problem easier. However, we do not restrict the stencil to just span one direction, but we allow communication along both axes. Formally, the stencil can be defined as $S = [(1,0),(-1,0),(0,1),(0,-1)]$. This problem is the same one as Gropp [7] attempted to solve in his work (restricted to two dimensions).

Previously, we could leverage the fact that the simple stencil only communicates along one axis, allowing for easier quality Q analysis. Now the optimal shape of a partition is no longer defined by a simple, long strip, but rather by a shape that resembles the most compact rectangle. Even worse, depending on the dimensions and partition sizes, the most optimal partition shape might not even be a rectangle. Unfortunately, it now becomes apparent that similar proof techniques will not work as easily as they did before.

What we are left with, is a problem with similar properties to common packing problems, such as the strip packing problem, which is known to be strongly NP-hard [16]. Lodi et al. [14] have compiled an excellent survey on common 2D-packing problems and compare various properties. Unfortunately, all these known problems mostly deal with rectangles or items of fixed, unmodifiable shapes. Even though our specific variation, where partitions may be reshaped if needed, does not necessarily mean that the problem becomes harder, all evidence so far point towards our problem instance also being at least NP-hard, if not even harder. The intuition behind this argument would be that more possible partition shapes must be regarded when finding a good mapping, ultimately enlarging the search space.

3.6 Problem Difficulty Assessment

In Section 3.2 we show that a restricted version of the problem is NP-hard. From this, we can derive that the unrestricted mapping problem also has to be NP-hard, since it partially consists of an NP-hard sub-problem, as proven with Theorem 3.2.1.

Corollary 3.6.1. *The unrestricted **GRID-PARTITIONING** problem is NP-hard.*

We also show that for equal sized partitions and the same component stencil, there exists an algorithm, which solves the problem in polynomial time. In Section 3.5 we argue that the addition of the 5pt stencil probably increases the difficulty, but could not prove NP-hardness.

Conjecture 3.6.1. *The **GRID-PARTITIONING** problem is NP-hard, when restricted to two dimensions, 5pt stencil and equal-sized partitions.*

From this point on, we will proceed under the assumption that the problem is already NP-hard for equal-sized partitions and 5pt stencil.

CHAPTER 4

Gropp's Algorithm

Due to the expected NP-hardness of the general problem, the focus of this work will now shift towards analyzing different heuristics. We will start by analyzing the existing proposition by Gropp [7], whose good experimental results were one of the main motivations for this thesis. Note that Gropp does not account for differently shaped stencils, but assumes square-like shapes. Therefore his solution does not correspond to the unrestricted **GRID-PARTITIONING** problem, but to the version, which is restricted to 5pt stencils and equal sized partitions.

4.1 Algorithm Description

Since a precise explanation of the algorithm can be found in Gropp [7, Section 2], this section will focus on the intuitive idea behind the approach. Keep in mind that this algorithm was designed to work for equally sized partitions and 5pt-stencils.

The idea is that a partition's communication performance corresponds to its mapping's circumference since these edges are responsible for communicating with other partitions. Gropp allows only rectangular partitions, so the partition's circumference decreases if the rectangle's side-lengths become more similar - in other words, more square-like. This is achieved by calculating a prime factorization for both the partition size and the grid's dimensions. The partition's prime factors (descending in size) are then assigned to the dimension, whose prime factorization can still accommodate the partition's prime factors. This approach guarantees that if there exists a good mapping, where every partition is shaped and oriented in the same rectangular shape, it will be found.

A complexity analysis for this algorithm is not easy. In short, the algorithmic complexity is bounded by the prime factorization, which is proven to be NP-hard and even expected not to be NP-complete [6]. Practically speaking, this factorization is done for usually small numbers, as partition sizes are (in this context) bounded by the core count of the machines in use and the dimension sizes are bounded by the number of machines in use.

Let us denote the number of prime factors of the partition size p with prime$_p$ with $|prime_p| \ll p$ and the dimensions' D prime factors as $prime_{D[i]}$ with $|prime_{D[i]}| \ll D[i]$. Further, let us denote the number of all dimensions' prime factors as $primes_D$ where $primes_D = \sum_{i=1}^{|D|} |prime_{D[i]}|$. Since every factor in $prime_p$ has to be compared to each dimension's factors, the overall runtime can be approximated with $\Theta(|prime_p| \cdot primes_D)$, when disregarding the runtime for the factorization completely.

4.2 worst-case Quality

While the algorithm's idea looks very promising, the generated mappings are not guaranteed to be optimal. This should not come as a surprise since the overall problem is expected to be NP-hard. In this sections we will show that Gropp's algorithm can become arbitrarily bad by showing a method for constructing worst-case instances of any scale. I will focus on 2D examples, but the procedure works for any number of dimensions - it even becomes more effective, with increasing dimension count.

The main drawback of the algorithm is that it relies on partition and dimension sizes to have well behaved prime factorizations. Let us now choose a partition size p, where p is a prime number. Gropp's algorithm, therefore, has no choice, but to assign the partition along one dimension in the shape of a 1-wide strip, since there are no other prime factors, except for p itself. This would lead to the rectangle's circumference being $2p + 2$. Assuming that a square-like shape is the most optimal for any partition, we can optimistically approximate an optimal solution's quality with the circumference of the square with the same surface area as the one-wide rectangle $4\sqrt{p}$. The ratio between these two qualities can be calculated with

$$\frac{2p+2}{4\sqrt{p}} = \frac{p+1}{2\sqrt{p}}$$

and we can observe that this ratio becomes infinitely bad, the bigger the partition size becomes. Gropp's algorithm, therefore, does not have a bound regarding any optimality guarantees.

$$\lim_{p \to \infty} \frac{p+1}{2\sqrt{p}} = \infty$$

Similarly, we can assume that one of the two (or both) partition sizes is a prime number. Wlog. let us assume that $D[1]$ is prime, which results in two case distinctions: $D[1] \in prime_p$ and a factor of $prime_p$ will be assigned to $D[1]$ or $D[1] \notin prime_p$. If $D[1] \in prime_p$ and a factor of $prime_p$ will be assigned to $D[1]$, then the partition will be at least $D[1]$ long. Similarly to before, as $D[1]$ becomes larger in comparison to p, the ratio to the optimal solution becomes worse. If $D[1] \notin prime_p$, then we know for sure again that all partitions will be shaped in 1-wide strips, as they cannot span $D[1]$, due to no common prime factors.

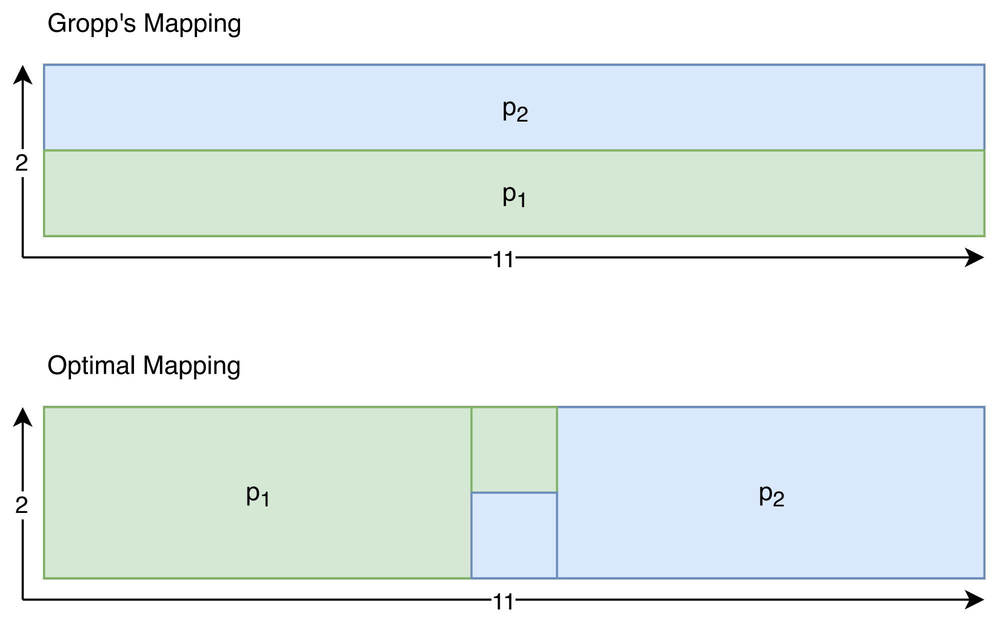


Figure 4.1: This exemplary mapping shows how Gropp's [7] algorithm performs when the partition sizes do not have well behaved prime factorizations. The other mapping is optimal wrt. a 5pt-stencil and both proposed optimality criteria.

A simple example of this construction can be seen in Figure 4.1 where the two partitions have a size of 11, meaning that they have to be arranged in 1-wide, 11-long strips. Note that the partitions do not necessarily have to be indivisible. For example, a partition size of 22 still forces the rectangles to be at least 2-wide strips, which can result in sub-optimal mappings as well. Another inherent drawback is that the algorithm was only designed for stencils that favour square-like partitions - in particular, 5- and 9-point-stencils. If the user wants to use other patterns, then the quality heavily depends on the stencil's asymmetry and may become worse.

CHAPTER 5

k-*d* Tree Algorithm

In the previous chapter, we presented Gropp's approach for solving the restricted version of the **GRID-PARTITIONING** problem, where only 5pt stencils and equal-sized partitions are allowed. To increase the applicability of our algorithm, we propose an algorithm for approximately solving the partitioning problem also for differently-sized partitions in this chapter. In addition to the algorithm description itself, a runtime and worst-case quality analysis will be given, similarly to the previous approach.

5.1 Algorithm Description

The idea for this algorithm is based on a data structure called k-d tree, which is useful when multidimensional data needs to be ordered or queried. It works similarly to a binomial tree, however, the decision variable, which is responsible for deciding whether to traverse the left or the right side, changes for each layer. Usually with k-d-trees, each layer corresponds to a different dimension, alternating in a round-robin like manner. In order to construct a more balanced tree, this implementation always finds the biggest dimension k, which is then split in half. The left side then only contains nodes with coordinates c for which $c[k] \leq \lfloor \frac{D[k]}{2} \rfloor$ and the right side contains the remaining nodes. Listing 5.1 shows the pseudocode for the algorithm which recursively halves the biggest dimension, until the desired coordinate is reached. Figure 5.1 shows a simple example with two dimensions $D[1] = 10, D[2] = 3$.

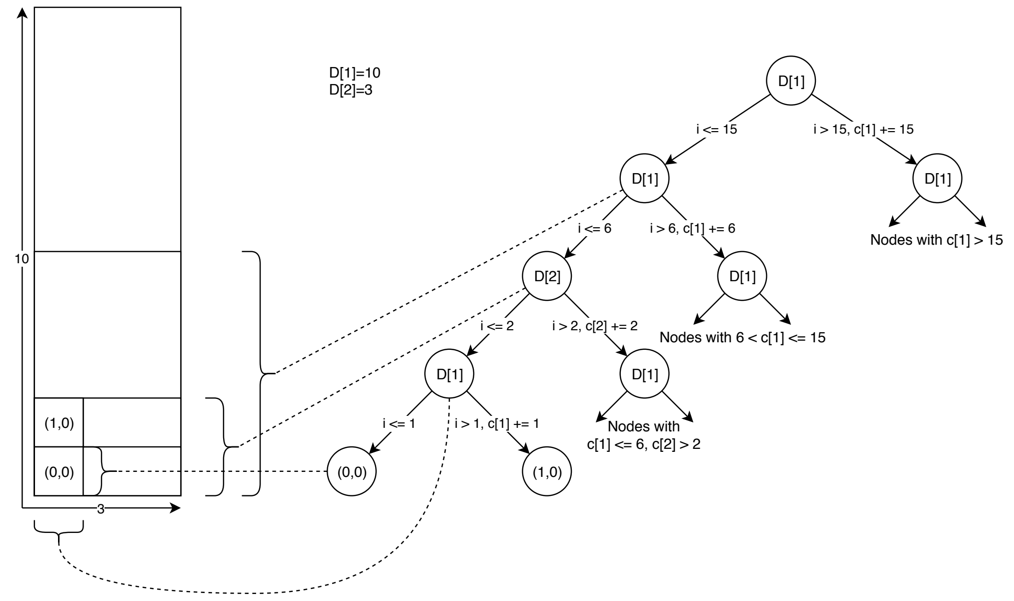


Figure 5.1: The traversal of a 10 by 3 grid can be seen here. Each side of the tree on every level corresponds to one half of the parent node's search space. It can be seen how node $(0, 0)$ can be found by always choosing the left side and $(1, 0)$ by choosing the right side once in the very end.

Listing 5.1: The pseudocode for the *k-d* tree algorithm's coordinate calculation. The simple recursion shows that the coordinate calculation is always done for the biggest dimension. Note that this exact approach modifies the dims array, so a copy should be made before.

```
void find_coordinates(
  int *dims     /* array dimensions */,
  int *coord    /* coordinates so far */,
  int elements /* number of elements in dims */,
  int target_index /* desired rank */
) {
  int k = index_of_largest_dim(dims);
  if (dims[k] <= 1) {
    return;
  }
  int left_size = (elements / dims[k]) * (dims[k] / 2);
  if (target_index < left_size) {
    dims[k] = dims[k] / 2;
    find_coordinates(dims, coord, left_size, target_index);
  } else {
    dims[k] -= dims[k] / 2;
    coord[k] += dims[k] / 2;
    target_index -= left_size;
    find_coordinates(dims, coord, elements - left_size,
      target_index);
  }
}
```

This algorithm requires all processes of the same partition to have an ascending target index. If not, then each process must calculate its correct target index first. In MPI this is usually achieved for free since the standard rank of `MPI_COMM_WORLD` is commonly already assigned that way. This may depend on the systems scheduler settings or on other environment factors.

Note that this algorithm therefore also works for differently sized partitions, since the only information needed for calculating the coordinate is the process' index in this ordered list - aka its MPI rank on the world communicator. This means that in a 10 by 10 grid, for example, the process with rank 17 will always get the same coordinate, regardless of how many partitions there are and what sizes they have. Additionally, all calculations may be done locally and no communication is required throughout the entire algorithm.

5.2 Runtime Analysis

Each iteration the currently biggest dimension has to be found (Line 7 in Listing 5.1). This may be improved by some priority queue, where the split dimension's new size may be re-appended, however, let us assume the cost to be linear wrt. $|D|$ in every iteration. The remaining calculations within the function are simple arithmetics and may be done in constant time.

The recursion ends when there is only one node within the current sub-tree. Since every iteration splits the biggest dimension in half (± 1 due to integer division), the number of remaining nodes in the search space is always halved. The k-d tree's depth, therefore, is limited to $\lceil \log_2(n) \rceil$, where n is the number of nodes. Again, a slight improvement may be to end the recursion whenever the `target_index` reaches 0, which would ultimately not change the complexity since the last rank has to be waited for anyway.

The overall complexity for calculating the rank's coordinate within the grid is $O(\log_2(n) \cdot |D|)$. Note that the necessary time for creating an actual MPI communicator is not regarded as being part of the algorithmic complexity.

5.3 worst-case Quality

For examining the quality wrt. the bottleneck partition, let us specify some arbitrary, but fixed properties. Let n denote the total number of processes, P denote the set of partitions, resulting in $|P|$ denoting the number of partitions and $P[i]$ the size of partition i. Similarly, let D denote the set of dimensions and $D[i]$ denote the size of dimension i. In the first split, the processes' indices are divided into the lower half and the upper half of the currently biggest dimension's indices. Let us denote this biggest dimension's index with k.

Since the algorithm itself does not directly account for partition sizes, three possible cases might have occurred. Firstly, no partition was split in two, which happens if the splitting index is the last element of a partition. This is the optimal case, but no concrete computations can be made yet for the bottleneck partition. Another case might be that a partition a was split, but one side is completely occupied by a part of a's processes and the other side is only partially occupied by a, while the rest might be filled with processes of possibly multiple other partitions. A small example for that would be a 10 by 10 grid with two partitions $a = P[1] = 55$ and $P[2] = 45$. The first split will divide the 10×10 grid into two sides, each containing 50 nodes. $P[1]$'s first 50 processes will occupy the left half completely, with the remaining 5 occupying parts of the second half. In this case, the bigger part a' of a will form a perfect hyperrectangle, with the number of nodes being equal to the volume of the hyperrectangle

$$V(D,k) = \frac{D[k]}{2} \cdot \prod_{i \in \{1..|D|\} \setminus k} D[i] = \frac{\prod_{i=1}^{|D|} D[i]}{2}$$

or more briefly, with a modified D' st. $D'[k] = \frac{D[k]}{2}$

$$V(D') = \prod_{i=1}^{|D'|} D'[i].$$

The number of outgoing edges are equal to the surface area of the hyperrectangle, namely

$$A(D,k) = (\prod_{i=1, i \neq k}^{|D|} D[i]) + (2 \sum_{i=1, i \neq k}^{|D|} \prod_{j=1, j \neq i, j \neq k}^{|D|} D[j])$$

or more briefly, with a modified D' st. $D'[k] = \frac{D[k]}{2}$

$$A(D') = 2 \sum_{i=1}^{|D'|} \prod_{j=1, j \neq i}^{|D'|} D'[j]).$$

The processes on the other side of the split point will now - by construction - be on the edge of future splitting operations. Let them be denoted by a''. This sub-partition a'' will now be subject to these two splitting cases in future - it either fills a hyperrectangle perfectly, terminating this recursive behaviour or it will be split again, where one side (the one on the edge of the previous splitting operation) will fully occupy a rectangle and the other side will have to share the space with other partitions. A key observation is that these recursively created sub-partitions will always be connected to the bigger partition on the other side of the split point.

The third case is a special case of the second one. A partition a is split, but both sides are not completely occupied by processes of a. This is the worst possible scenario since the two sub-partitions a' and a'' might not even be connected anymore, increasing the number of outgoing connections drastically. However, these two partitions are now certainly at the edge of future split points, similarly to the behaviour in the previous case. Therefore, this third case, which completely separates two partitions, may only occur once per partition.

Per hyperrectangle, the worst possible ratio of outgoing edges versus the number of nodes can be achieved by maximizing $\frac{A(D,k)}{V(D,k)}$, which can be done by skewing the dimensions' ratio and by lowering the number of dimensions. This is not surprising, as intuitively a long, 1-dimensional strip has a worse surface to volume ratio than an equal-sided 3-dimensional (or n-dimensional) cube.

5.3.1 Smallest Partition to fill any Rectangle

The worst possible ratio of a whole partition is then made up by maximizing the area of all sub-partitions' rectangles and the partition's size $\frac{\sum_{a' in a} area(a')}{|a|}$. This may be achieved by creating a polygon within the rectangle that has the maximum surface area with the smallest volume. The fact that obviously differently sized polygons can have the same perimeter can be seen in Figure 5.2, where all of the shown, green figures within the rectangle have the same surface area.

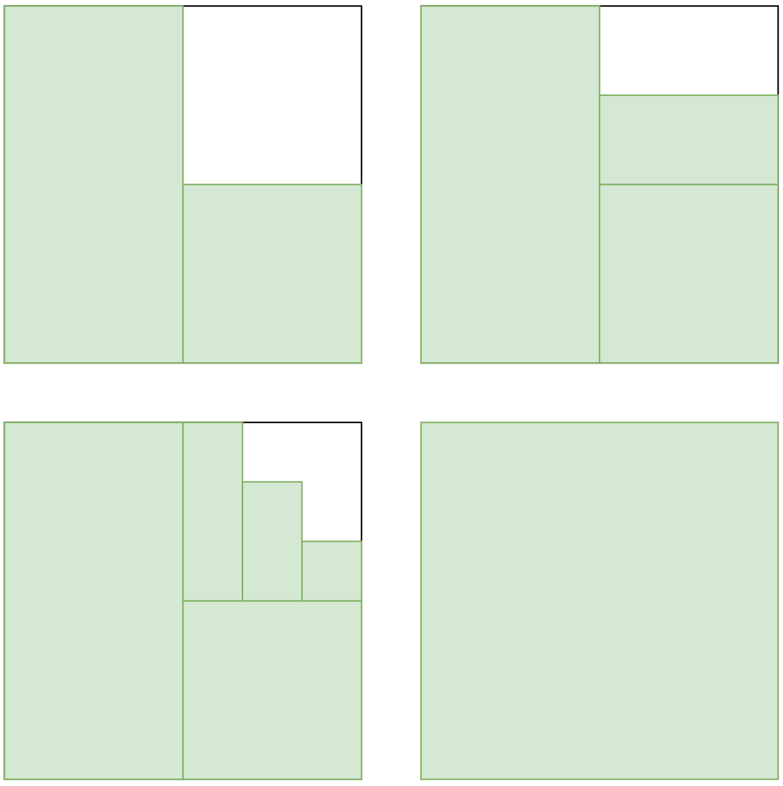


Figure 5.2: All these green polygons have the same surface area despite having different volumes.

The key to achieving the smallest polygon is to attempt to span every dimension completely whenever a split happens. In other words, at every split, we ask the question: "Do we need to go into the right side of the tree, in order to fill the rectangle?". Figure 5.3 visualizes this procedure.

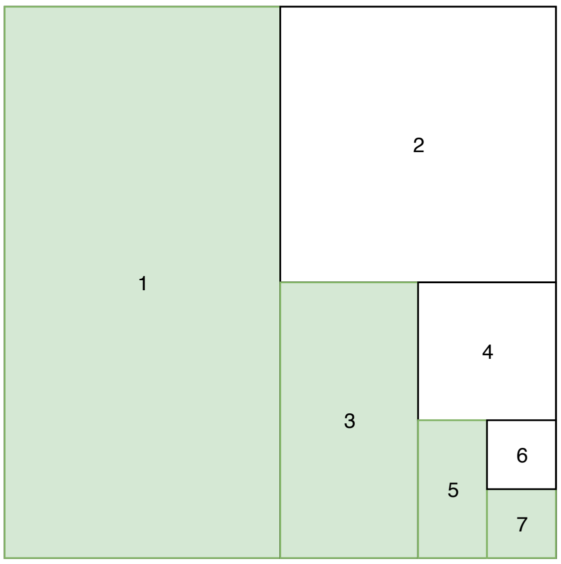


Figure 5.3: The procedure for deciding the required sub-rectangles when trying to achieve the smallest number of nodes.

The first step has to be handled carefully since we always only want to take one half of the rectangle (Rectangle 1). When the first rectangle is chosen, we already cover all dimensions, except for the largest one, where we made the cut. After the second split, we can see that this one cuts across another dimension, meaning that attempting to reach for rectangle 2 would only give us the possibility to cover dimensions that we already have covered anyway. We, therefore, chose not to overextend and continue splitting the bottom rectangle. After the third split, we cut across our original dimension again, meaning that we need to cover the right half of the decision tree as well. This procedure goes on recursively until the last, single element has been picked. Note that this algorithm behaves similar for an arbitrary number of dimensions - only the original splitting dimension is the one, where the right side of the tree is chosen.

The number of covered nodes therefore depends on when the cuts across the original dimension happen. Let us suppose that cuts happen across the first dimension three times in a row, then we at least need $\frac{1}{2} + \frac{1}{4} + \frac{1}{8} = \frac{7}{8}$ nodes already - compared to only $\frac{1}{2} + \frac{1}{8} = \frac{5}{8}$ when the second cut is across another one. However, we can prove that once another dimension was halved, the original dimension k and i will be split in alternation. Let us suppose that the last split, where k was halved was at iteration x, then we have

$$\frac{D[k]}{2^x} \geq D[i] > \frac{D[k]}{2^{x+1}}.$$

In the next iteration $D[i]$ will be halved, leaving us with $2D'[i] = D[i]$.

$$\frac{D[k]}{2^x} \geq 2D'[i] > \frac{D[k]}{2^{x+1}} = \frac{D[k]}{2^{x+1}} \geq D'[i] > \frac{D[k]}{2^{x+2}}$$

We now know that our previous remainder of $D[k]$ after x splits is now larger again than the newly created $D'[i]$, proving that the splitting will now alternate between the two dimensions. We can compute after which iteration this behaviour will occur, by solving the equation for x:

$$\frac{D[k]}{2^x} \geq D[i] > \frac{D[k]}{2^{x+1}} \;=\; \log_2\left(\frac{D[k]}{D[i]}\right) \geq x > \log_2\left(\frac{D[k]}{D[i]}\right) - 1 \quad \Longrightarrow \quad x = \log_2 \frac{D[k]}{D[i]}$$

In order to find out, what happens when another dimension becomes smaller than both $D[k]$ and $D[i]$, we could modify the formula again, but that is beside the point. We could successfully illustrate that a heavily skewed rectangle increases the number of nodes required to fill it. If it were a perfect hyper-square, then the alternation between splitting dimensions would start immediately, minimizing the number of nodes required.

We can calculate the number of nodes n needed since we know how often the rectangle is halved in one dimension. The remaining space is then split, alternating between both dimensions a and b. Wlog. we can assume that $a > b$.

$$x = \sum_{i=1}^{ld(\frac{a}{b})} \frac{1}{2^i}, \quad y = \sum_{i=0}^{\frac{ld(b^2)}{2}} \frac{1}{2} \cdot \frac{1}{4^i}, \quad n = (x + (1 - x) \cdot y) \cdot ab$$

where x denotes the number of nodes that will be reserved until the splitting will alternate between a and b and y denotes the splitting of the square-like part of the rectangle. However, this formula is hard to work with and becomes even more complicated for multiple dimensions. Since we are only interested in the worst-case scenario, we will not calculate the minimum required nodes exactly, but just approximate it. There are two aspects that come into play here. Firstly, the fact that we always need at least one half of the rectangle and secondly that the second half will be iteratively split between the participating dimensions at some point. However, since we are working with integer numbers, this repetitive quartering is not so straight forward to anticipate, which is why we will pessimistically ignore this part. The resulting approximation of the number of required processes is

$$n = \frac{ab}{2}.$$

Now that we can approximate the smallest number of processes, which are required to fill the given hyperrectangle, we can perform the worst-case analysis. Note that the acquired bound will not be tight, because of the relaxations we made.

5.3.2 Unconstrained Worst-Case

For the absolute worst-case, we would need to introduce an example, where the third case, which splits the partition in the middle, happens. Let us say that the two sub-partitions are completely disconnected and look at an example for just two dimensions. For one side of the split, we know that the area is $2a + 2b$ and the volume to be at least $\frac{ab}{2}$. For two disconnected rectangles we, therefore, have a total area of $4a + 4b$ and ab nodes. If these ab could be arranged optimally, then they would form a perfect square with the four sides having the length $4 \cdot \sqrt{ab}$. The optimality ratio between our algorithm's solution and the potential worst-case therefore would be:

$$\frac{4a + 4b}{4 \cdot \sqrt{ab}} = \frac{a + b}{\sqrt{ab}}$$

Since we assumed wlog. that $a \geq b$, we can express a as $a = b \cdot s$ where s denotes the rectangle's skew, which has to be greater or equal to 1.

$$\frac{bs + b}{\sqrt{bs \cdot b}} = \frac{s + 1}{\sqrt{s}}$$

What we can see here is that our optimality ratio becomes worse with an increasing skew s. In the best case of $s = 1$, the resulting ratio would be 2, meaning that the worst possible solution is two times worse than what could be possible in the optimal case. Note that this bound is not tight, since we completely omitted assigning the right half of the rectangle.

We can modify this formula to accommodate an arbitrary amount of dimensions as well. d_i will denote the ith rectangle's side length and n the number of dimensions.

$$\frac{4 \sum_{i=1}^{n} \prod_{j=1, j \neq i}^{n} d_j}{2n \sqrt[n]{\prod_{i=1}^{n} d_i}^{n-1}} = \frac{2 \sum_{i=1}^{n} \prod_{j=1, j \neq i}^{n} d_j}{n \sqrt[n]{\prod_{i=1}^{n} d_i}^{n-1}}$$

For example, when $n = 3$, $d_1 = 10, d_2 = 8, d_3 = 6$ then we would get a ratio of

$$\frac{2 \cdot 8 \cdot 6 + 2 \cdot 10 \cdot 6 + 2 \cdot 10 \cdot 8}{3 \sqrt[3]{10 \cdot 8 \cdot 6}^2} = \frac{376}{183.914...} = 2.044...$$

5.3.3 Constrained worst-case

Previously we saw that the more the hyperrectangle is skewed, the bigger the optimality ratio becomes. However, what would it mean for the largest rectangle's side d_k to be more than a factor 2 off of any other dimension d_i? Since the procedure only ever halves the largest dimension, d_i cannot have been halved even once yet. If it would have been, then $2d_i \geq d_k$ must have held at some point, but we just assumed that $d_k > 2d_i$. $2d_i > 2d_i$

clearly contradicts our assumption, meaning that the rectangle's current dimension i must be equal to what we started with $D[i]$. This, in turn, means that our optimal solution cannot be a square with a sidelength bigger than d_i, meaning that our solution's d_i dimension is in fact optimally chosen (as big as it can go).

What does that mean for our worst-case scenario? The biggest skew any dimension can have, compared to the biggest dimension is between 1 and 2. Again, wlog., let us assume that d_1 is the biggest dimension, then in the worst-case $\forall_{i \in \{2..n\}} d_1 = 2d_i$ holds. We can now simplify the general ration formula to match this worst-case:

$$\frac{2\left(\left(\frac{d_1}{2}\right)^{n-1} + (n-1)d_1 \cdot \left(\frac{d_1}{2}\right)^{n-2}\right)}{n \sqrt[n]{d_1 \left(\frac{d_1}{2}\right)^{n-1}}^{n-1}} = \frac{2\left(\frac{d_1}{2}\right)^{n-1}(1 + 2(n-1))}{n\left(\frac{d_1}{2}\right)^{n-1} \sqrt[n]{2}^{n-1}} = \frac{4n-2}{n\sqrt[n]{2}^{n-1}}$$

The resulting formula can be used to calculate the bound for the worst possible optimality ratio of the algorithm. For $n = 2$ that would be $\frac{3}{\sqrt{2}} = 2.12...$ and 2.09... for $n = 3$. This number converges towards 2 for increasing dimensions.

While this algorithm could improve upon Gropp's [7] restrictions by working for any partition size and having a bounded worst-case, it still favours square-shaped hyperrectangles. Optimal communication patterns are therefore limited to 5- or 9pt-stencils again. In an attempt to alleviate this problem, we propose a slight change to this *k-d* tree algorithm, in the following section.

5.4 Skewed *k-d* Tree Algorithm

The previously presented *k-d* tree algorithm was designed with 5-point stencils in mind. The quality of any solution may, therefore, become arbitrarily bad if the stencil, which was chosen by the user, differs from the ideal. Instead of aiming for square-like hyperrectangles during the splitting phase, the partitions' target shape will be influenced by the given stencil.

In line 7 of the previous pseudo code in Listing 5.1 we can see that the split-dimension is chosen solely by the remaining size of the dimension. Instead, we will introduce another function that weights the dimensions' remaining size by the inverse of how often the dimension is communicated along. Listing 5.2 shows the code for calculating the index of the largest, weighted dimension.

Listing 5.2: This function replaces the previously trivial method of finding the biggest dimension. The `factors` array contains the weight for each dimension, which is then inversely used, so that dimensions are preferred if there is less communication across them.

```c
int skewed_max_index(
  const int *dims,
  int num_dims,
  int stencil_size,
  int *stencil
) {
  double *factors = malloc(num_dims * sizeof(double));
  double m = -1;
  int max_index = -1;
  double skewed_entry;

  /* calculate each dimension's weight */
  for (int d = 0; d < num_dims; d++) {
    factors[d] = 0;
    for (int i = 0; i < stencil_size; ++i) {
      /* if the stencil points somewhere in that direction */
      if (stencil[(i * num_dims) + d] != 0) {
        factors[d]++;
      }
    }
  }

  for (int i = 0; i < num_dims; i++) {
    skewed_entry = dims[i];
    if (skewed_entry > 1) {
      /* always prefer to split dims wo. communication */
      if (factors[i] == 0) {
        skewed_entry = INT_MAX;
      } else {
        skewed_entry /= factors[i];
      }
    } else {
      skewed_entry *= -1;
    }
    if (skewed_entry > m) {
      m = skewed_entry;
      max_index = i;
    }
  }
  return max_index;
}
```

The recursions termination condition is only slightly adapted since the new function returns -1 if all dimensions are already of size 1 and cannot be split any further. The rest of the algorithm, namely the splitting and traversing of the tree, remains unchanged. The algorithmic complexity, therefore, is not impacted negatively, even though better results may be achieved for some stencils. Note, however, that wildly asymmetric stencils still will not be mapped optimally, since the targeted shape remains a hyperrectangle.

CHAPTER 6

Squarest Strips Algorithm

During the implementation of the *k*-*d*-tree algorithm, we found some small, disheartening examples where the standard behaviour of MPI found better solutions than both Gropp's algorithm or the *k*-*d*-tree approach. Figure 6.1 shows a very simple example where it can be seen that the *k*-*d*-tree algorithm creates jagged edges due to the recursive splitting in the two left-most partitions. MPI's standard behaviour, on the other hand, finds the optimal solution for this instance. The approach that we present in this chapter was inspired by this example. Note that the aim of this solution is to find square-like partitions again, therefore, this algorithm solves the restricted **GRID-PARTITIONING** problem for 5pt stencils.

6.1 Algorithm Description

Before describing the algorithm, we want to analyze, why the standard behaviour of MPI performs so well for these instances. The partitions from the example of Figure 6.1 are all of size 11. If they would be arranged in a perfect square, then the side length of said square would be $\sqrt{11} = 3.316...$. We only have integers to our disposal, so the side lengths have to be between 3 and 4. A 3×4 rectangle would contain 12 processes, which is pretty close to our partition size of 11. MPI's standard behaviour assigns processes starting with the first dimension (row-major). In this case, this dimension is only 4 wide, which coincides with one of the side-lengths of a pretty good bounding rectangle. The resulting partitions are therefore arranged quite compactly with only one to two jagged corners. Obviously, this approach only works if the grid size corresponds to the optimal side-length of the bounding rectangle, which is why the standard behaviour only performs so well for these small instances.

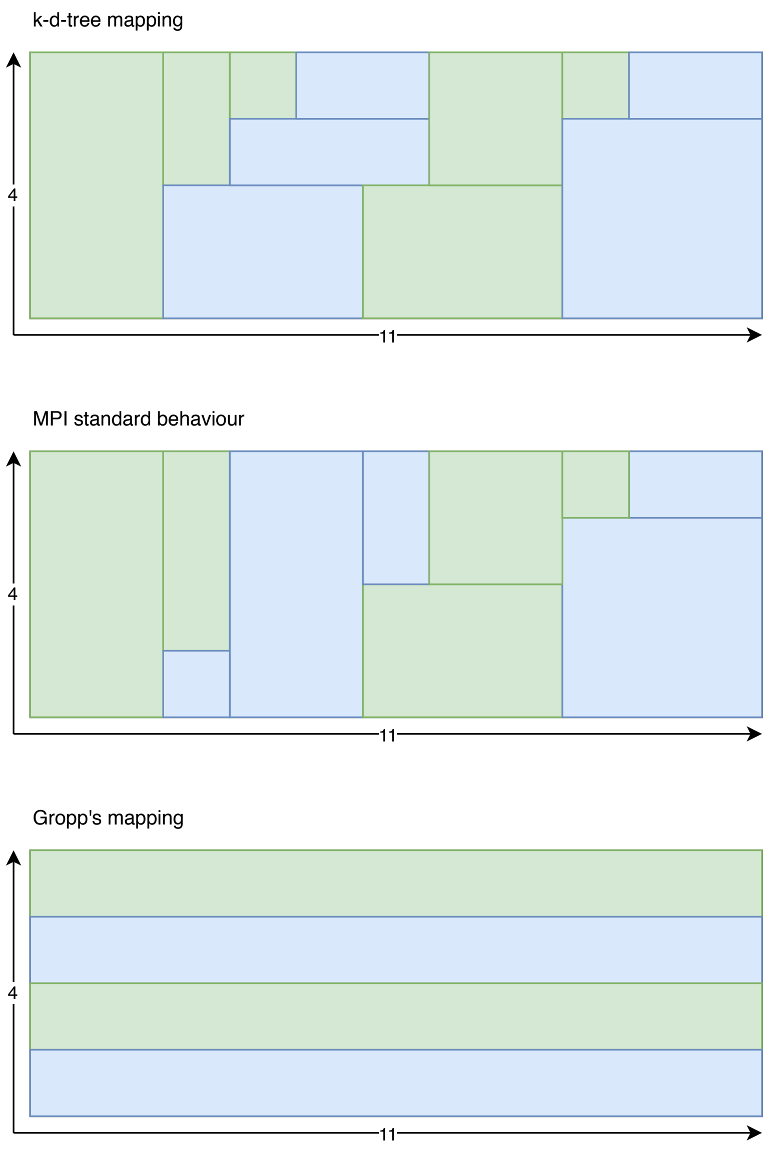


Figure 6.1: A simple 4 × 11 grid with 4 partitions, where three different mappings can be seen (Gropp, *k*-*d*-tree and the standard behaviour of MPI). Note how the standard behaviour is better wrt. the bottleneck partition than both other approaches.

6.1.1 Variation with Disjoint Partitions

In this section, the algorithm's concept will be explained with the help of a simple example. In its basic form, the squarest strips algorithm may produce multiple disjoint partitions, which is no good. The final, more complicated version requires some adaptations and will be presented in the following section, however, the core idea stays the same and will be discussed here.

For the n-dimensional case, the squarest strips algorithm works by calculating the nth root of the partition size p as the target width of a single strip. The first dimension is then divided by this target width, which gives us the number of strips within this dimension. Since neither the dimension has to be divisible by the nth root nor does the root have to be an integer, the actual strip width is then calculated. This width is then assumed to be one of the bounding rectangles side-lengths. The following dimensions' strip widths are then not calculated using the nth root, but rather using the already calculated, actual strip widths in mind. The formula for the ith strip width can be expressed as

$$\text{strip\_width}(i) = \sqrt[n-i]{\frac{p}{\prod_{j=0}^{i-1} \text{strip\_width}(j)}} \quad \forall i \in \{0 \dots (|D|-1)\}.$$

This calculation is done for every dimension, except for the largest one (The actual implementation is not recursive, but iterative and linear in time wrt. the number of the dimensions). This largest dimension is chosen to be iterated along, similarly to how the standard approach iterates over the grid in a row-major fashion.

Similarly to the k-d-tree approach, every process can locally compute its new rank, since the current rank corresponds to the index of the coordinate in the new grid. This can be done by iterating over all dimensions and calculating the strip width. We then also know how many processes fit into each strip, meaning that each process can compute to which strip it belongs. For an easier explanation, let us look at an example for calculating the coordinate of process 70 if there is a 3D grid of size $5 \times 4 \times 8$ and a partition size of 16. We know that for the first dimension one strip is 2 high ($\sqrt[n]{p} = \sqrt[3]{16}$) and the other 3. The first strip of height 2 contains $2 \cdot 4 \cdot 8 = 64$ processes, which means that our process 70 falls into the second strip, where it is the $70 - 64 = 6$th one. The second dimension also has two strips, both having a size of 2. These strips contain $3 \cdot 2 \cdot 8 = 48$ nodes each, which means that strip 1 is sufficient for fitting rank number 6. As already mentioned, the assignment across the last dimension is handled differently. In order to be able to re-use the algorithm, a strip width of 1 is assumed here, meaning that all 8 strips have the same dimensions of $3 \cdot 2 \cdot 1$ and therefore contain 6 processes each. Our process with rank 6 (0 based) is therefore in the second strip, where its new rank would be 0.

We now know that the process, which originally had rank 70 falls within the 2nd strip along the first, the 1st strip along the second and the 2nd strip along the third dimension, where it has a new relative rank of 0. Within this strip, all that is left to do is to assign a coordinate in the same way as the default behaviour would have done - row-major, however with the rows not being the dimension, but the strip dimensions.

After doing so, we now know that the coordinate assignment is rather compact within one strip, since the standard behaviour always operates within an appropriately sized bounding rectangle. However, every partition that spans more than one strip may be split into two components. Let us assume that in our previous example, our rank 70 is in the same partition as rank 60 - after all, the assumed partition size was 16. These two processes would be in two completely different strips, which do not even share one common edge. This issue will be addressed in the following section, where a slight adaptation guarantees partition connectedness.

6.1.2 Connected Partitions

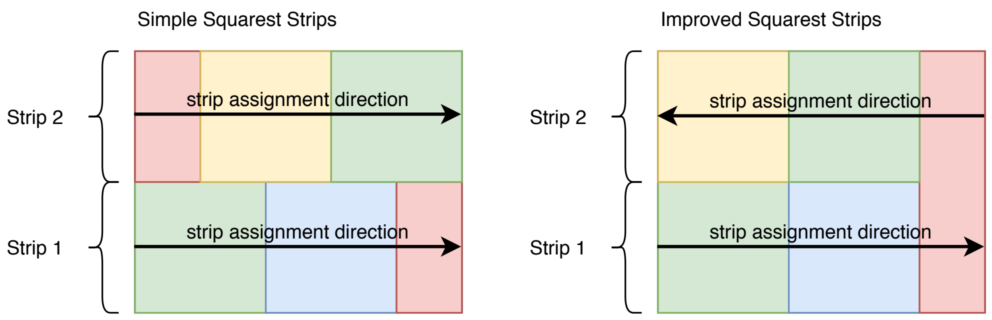

Simple Squarest Strips
Strip 2
Strip 1
strip assignment direction
strip assignment direction
Improved Squarest Strips
Strip 2
Strip 1
strip assignment direction
strip assignment direction

Figure 6.2: A simple example mapping where the simple squarest strips algorithm would split one partition in two. Due to the adapted assignment direction, the improved version does not do so.

The squarest strips algorithm in its final form is presented in this section and addresses some of the problems that could be seen previously, which can be seen in Figure 6.2. For the two dimensional case, we can see that the assignment direction may simply be corrected by alternating it every strip. However, for the three or n dimensional case, the setting becomes more difficult, as can be seen in Figure 6.3. Each strip's assignment direction depends on the previous strip choice.

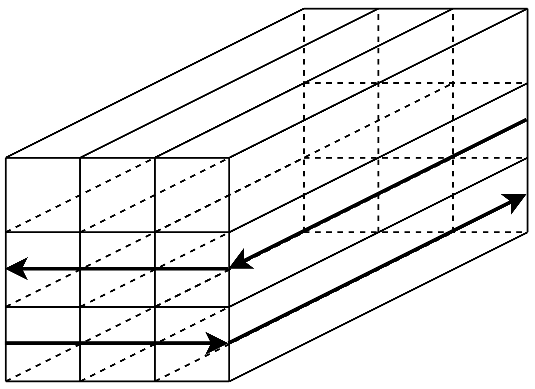


Figure 6.3: For more than two dimensions it becomes harder to keep track of repeated assignment-direction changes, since one depends on the other.

Listing 6.1: The pseudocode for the squarest strips algorithm shows the convoluted process of finding connected strips.

```
compute_strip_coordinate(/* args */) {
  bool flip_next = false;
  /* wlog. the last is the largest dimension */
  for (int curr_dim = 0; curr_dim < num_dims; curr_dim++) {
    int small_strip_height = /* nroot plus remainder */;
    int num_small_strips = /* number of "normal sized" strips */;

    // number of nodes in a single row within one strip
    int nodes_per_row = total_nodes / current_dim_size;
    int big_strip_nodes = ((small_strip_height + 1) * num_big_strips * nodes_per_row);
    int small_strip_nodes = total_nodes - big_strip_nodes;

    if (!flip_next) {
      // assuming that we start by filling the bigger strips
      bool small_strips_required = target_rank >= big_strip_nodes;

      if (small_strips_required) {
        // we know for sure that our rank surpasses all big strips
        target_rank -= big_strip_nodes;
        // calculate in which small strip we are
        int small_strip_index = target_rank / (small_strip_height * nodes_per_row);
        target_rank -= small_strip_index * small_strip_height * nodes_per_row;
        coord[curr_dim] += /* surely occupied strips */;
        strip_coord[curr_dim] = num_big_strips + small_strip_index;
      } else {
        // we know for sure that our rank is somewhere within the big strips
        int big_strip_index = target_rank / ((small_strip_height + 1) * nodes_per_row);
        target_rank -= big_strip_index * (small_strip_height + 1) * nodes_per_row;
        coord[curr_dim] += /* surely occupied strips */;
        strip_coord[curr_dim] = big_strip_index;
      }
    } else {
      // start by filling the smaller strips, since the direction is flipped
      bool big_strips_required = target_rank >= small_strip_nodes;

      if (big_strips_required) {
        // we know for sure that our rank surpasses all small strips
        target_rank -= small_strip_nodes;
        // calculate in which small strip we are
        int big_strip_index = target_rank / ((small_strip_height + 1) * nodes_per_row);
        target_rank -= big_strip_index * (small_strip_height + 1) * nodes_per_row;
        coord[curr_dim] += /* surely occupied strips */;
        strip_coord[curr_dim] = (num_big_strips - (big_strip_index + 1));
      } else {
        // we know for sure that our rank is somewhere within the small strips
        int small_strip_index = target_rank / (small_strip_height * nodes_per_row);
        target_rank -= small_strip_index * small_strip_height * nodes_per_row;
        coord[curr_dim] += /* surely occupied strips */;
        strip_coord[curr_dim] = num_big_strips + (num_small_strips - (small_strip_index + 1))
      }
    }
    if (strip_coord[curr_dim] % 2 == 1) {
      // alternate the assignment for the next strip, if this one was odd
      flip_next = !flip_next;
    }
    total_nodes = (total_nodes / dims[curr_dim]) * strip_sizes[curr_dim];
  }
  return coord and strip_coord;
}
```

The assignment direction not only impacts the last phase, where the row-major coordinate calculation happens, but also the strip choices themselves. Additionally, for each dimension, not every strip must have the exact same width, meaning that the whole assignment process has to account for either starting from the side, where the bigger strips are located or the side where the smaller strips are located. This rather convoluted process can be seen in Listing 6.1. The `flip_next` boolean is responsible for deciding where the strip assignment starts from. The same boolean is later used for the row-major assignment, so each iteration's value is stored in an array for later use (not shown in pseudo-code). The `strip_coord` array contains the indices of the strips themselves, not the process coordinates directly, additionally, all strip dimensions are kept track of in another array (not shown in pseudo-code). After the strip coordinate has been found, another function is responsible for calculating the final coordinates, based on the strip's dimensions and `flip_next` array, shown in Listing 6.2.

Listing 6.2: The processes relative position within a strip is added to the already known strip coordinate.

```
compute_coord(int target_rank,
              const int *strip_dimensions,
              int *coord, const bool *flipped_dim) {
  for (int curr_dim = 0; curr_dim < num_dims; curr_dim++) {
    int coord_component = target_rank;
    int strip_size = strip_dimensions[curr_dim];

    for (int rem_dim = num_dims-1; rem_dim > curr_dim; rem_dim--) {
      coord_component /= strip_dimensions[rem_dim];
    }

    if (flipped_dim[curr_dim]) {
      // the numbering starts from the strip's end
      coord[curr_dim] += strip_size - ((coord_component % strip_size) + 1);
    } else {
      coord[curr_dim] += coord_component % strip_size;
    }
  }
}
```

6.2 Runtime Analyisis

As already shown in Listing 6.1, the main procedure loops over each dimension exactly once. Finding the largest dimension and modifying the dimension array accordingly was also done linearly wrt. the number of dimensions. All hidden computation, such as the calculation of the base strip height, was done in constant time. Since the for loop contains no recursive elements, this procedure runs in $O(|D|)$. With a few modifications to the code, many arrays, including the coordinate and the strip coordinate, can be replaced by directly modifying the target rank. This reduces the real-world runtime and memory usage, but does not change the theoretical complexity. The final coordinate computation, which was shown in Listing 6.2, also loops over all dimensions. Within each loop, however, the remaining dimensions are also iterated over, increasing the theoretical runtime to $O(|D|^2)$.

This is an important improvement over all previously presented algorithms in this thesis. The runtime is independent of the number of processes, the number of partitions, the partition sizes and of the individual dimension sizes. Due to the way it was implemented, it only depends on the number of dimensions.

6.3 worst-case Quality

Regarding the solution's quality, we have to consider two factors. The first one is how close a strip's dimensions actually come to being a perfect hypercube. Secondly we have to consider the impact of partitions whose processes span over multiple strips. Similarly, to previous chapters, we will only consider the bottleneck partition here. Recall the recursive definition of the strip width calculation as

$$\text{strip\_width}(i) = \sqrt[n-i]{\frac{p}{\prod_{j=0}^{i-1} \text{strip\_width}(j)}} \quad \forall i \in \{0 \ldots (|D|-1)\}.$$

However, this is only the base strip width, disregarding the fact that this number is not an integer and the remaining nodes still have to be distributed. For example, in two dimensions and a partition size of 16, the base strip width of the first strip should be $\sqrt[2]{16} = 4$. However, if the first dimension is 7 long, then there will only be space for one strip, instead of two. This one strip will then span the whole dimension and have a width of 7. We can modify the formula accordingly to accommodate this change.

$$s(i) = \left\lceil \frac{d_i}{\left\lfloor \frac{d_i}{\sqrt[n-i]{\frac{p}{\prod_{j=0}^{i-1} s(j)}}} \right\rfloor} \right\rceil$$

The $(n-i)$th root still represents the desired strip width. By dividing the actual dimension size by this value, we get the number of strips that will be present in this dimension (rounded down). By doing seemingly the same division again, we get the actual strip width as indicated by the number of strips present, instead of the desired strip width. This value may still be off by 1 since some strips may be a little bit larger than others, so the result is rounded up. After calculating all strip widths, we need to find out how far this bounding rectangle may span across the last, largest dimension. Coincidentally, this calculation is equivalent to the calculation which we already performed in the denominator of the previous calculation.

Since we want to calculate the worst possible outcome regardless of specific dimensions, we will relax this formula to be more pessimistic. We will assume that every strip height is actually twice as big as it would optimally be (minus one). This gets rid of integer divisibility issues and leaves us with the following formula.

$$s'(i) = \begin{cases} 2 \cdot \left\lfloor \sqrt[n-i]{\frac{p}{\prod_{j=0}^{i-1} s'(j)}} \right\rfloor - 1, & \text{if } i < |D| - 1 \\ \left\lceil \frac{p}{\prod_{j=0}^{i-1} s'(j)} \right\rceil + 1, & \text{if } i = |D| - 1 \end{cases}$$

We can now express the surface area of the bounding rectangle as

$$A(rect) = 2 \sum_{i=0}^{|D|-1} \prod_{j=0, j \neq i}^{|D|-1} s'(j)$$

and the optimality ratio between this area and a perfect hypercube as

$$\frac{2 \sum_{i=0}^{|D|-1} \prod_{j=0, j \neq i}^{|D|-1} s'(j)}{2n \sqrt[n]{p}^{n-1}} = \frac{\sum_{i=0}^{|D|-1} \prod_{j=0, j \neq i}^{|D|-1} s'(j)}{n \sqrt[n]{p}^{n-1}}.$$

Since the non-trivial, recursive definition of s' makes it hard to find the worst-case scenario for our algorithm, Table 6.1 shows the numerical evaluations for partitions with a size of up to 32 and 10 dimensions. It can be seen that all of these values are below 1.9, which is a theoretical improvement over the *k-d* tree algorithm with a quality bound of ~ 2.12.

In order to formally calculate an optimality bound, we could further relax this expression to be non-recursive. The recursion was implemented in order to balance out strip widths, which might have been skewed in previous dimensions. By removing this behaviour, we can then assume that the strip has twice the optimal width on every dimension, which can be expressed as

$$2 \sqrt[n]{p} - 1.$$

Accounting for the last dimension's strip width, this results in a total bounding hyper-rectangle area of

$$2n \left(2 \sqrt[n]{p} - 1\right)^{n-2} \left\lceil \frac{p}{\left(2 \sqrt[n]{p} - 1\right)^{n-1}} \right\rceil < 2n \left(2 \sqrt[n]{p}\right)^{n-2} \left(\frac{p}{\left(2 \sqrt[n]{p}\right)^{n-1}} + 1 \right).$$

Table 6.1: This table shows the numerically evaluated worst-case scenarios for the squarest strips algorithm, when compared to the area of a perfect hypercube. Note that all values are below 1.9.

p	2 dims	3 dims	4 dims	5 dims	6 dims
2	1.4142	1.4699	1.4865	1.4933	1.4966
3	1.7321	1.7627	1.7548	1.7440	1.7347
4	1.1667	1.2787	1.2964	1.2975	1.2949
5	1.1926	1.4060	1.4455	1.4533	1.4530
10	1.2649	1.1730	1.2152	1.2151	1.2069
12	1.2702	1.2083	1.3184	1.3424	1.3450
14	1.2829	1.2433	1.4047	1.4450	1.4539
16	1.3214	1.2774	1.4792	1.5307	1.5433
18	1.3132	1.3104	1.5448	1.6043	1.6189
20	1.3097	1.3421	1.6037	1.6689	1.6842
25	1.3556	1.3126	1.7292	1.8021	1.8164
30	1.3389	1.3465	1.1702	1.2241	1.2339
32	1.3356	1.3603	1.1892	1.2625	1.2807

However, this simplification already assumes the calculated strip widths to be so wrong, that it would not make sense to continue this calculation. It would lead to an optimality bound of $O(2^n)$, which is not representative. For example, let us assume a 3D grid and a partition size p of 27. This simplification assumes the first two dimensions' strip widths to both be 6, resulting in the last dimension's strip width of 1. In actuality, the worst-case would result in a maximum strip width of 5 for the first, 3 for the second and 2 for the last dimension, which is a much better result even in the worst-case scenario. For the interested reader, we present this formula nevertheless:

$$\frac{2^{n-1} n \sqrt[n]{p}^{n-2} + n \sqrt[n]{p}^{n-1}}{2n \sqrt[n]{p}^{n-1}} = \frac{1}{4}\left(2^n p^{-1/n} + 2\right)$$

The second aspect we want to consider is the impact of partitions whose processes span over multiple strips. We could already see in Figure 6.2 that the improved version of the algorithm keeps the partitions connected, but they are still stretched. The stretching dimension is never the last one since it is responsible for iterating over within a strip. Let us suppose there is no periodicity across the edge, where the partition has to stretch along. In that case, one of the partition's two stretched edges does not contribute to any inter-node communication. This property actually reduces the number of inter-node edges. Let us look at the following formula of the hyperrectangle's surface area, where k denotes the dimension, which is stretched.

Table 6.2: The numerically evaluated worst-case scenarios for the squarest strips algorithm, when a partition might stretch across a second strip. The comparison here is between the *non-periodic* case and the perfect hypercube. Since one edge of the hypercube is subtracted, it might occur that the ratio for this stretched case is better than for the non-stretched ratio, since no edge is subtracted for the perfect hypercube.

p	2 dims	3 dims	4 dims	5 dims	6 dims
2	1.0607	1.4699	1.6352	1.7230	1.7772
3	1.4434	2.0832	2.3031	2.4084	2.4686
4	1.1667	1.4992	1.8856	2.0672	2.1699
5	0.9690	1.1020	1.3209	1.4165	1.4675
10	1.1068	1.4363	1.5412	1.7962	1.9244
12	1.1258	1.5263	1.7061	2.0273	2.1856
14	1.1492	1.0903	1.1168	1.2674	1.3368
16	1.1964	1.2074	1.5938	1.5162	1.6150
18	1.1953	1.3104	1.8023	1.7232	1.8437
20	1.1979	1.4024	1.9826	1.8995	2.0366
25	1.2556	1.1566	2.3479	2.2488	2.4130
30	1.2476	1.4155	1.5992	1.6585	1.8803
32	1.2473	1.4706	1.6475	2.0333	1.9674

$$s''(i) \begin{cases} 2 \cdot s'(i), & \text{if } i = k \\ s'(i) - 1, & \text{if } i = |D| - 1 \\ s'(i), & \text{otherwise} \end{cases}, \quad A'(rect) = \left(2 \sum_{i=0}^{|D|-1} \prod_{j=0, j \neq i}^{|D|-1} s''(j) \right) - \prod_{i=0}^{|D|-2} s''(i)$$

The left term in A' is similar to the unstretched formula, where both sides on the stretched dimension are accounted for. The right term then removes the nodes, which are guaranteed to lie on an edge. In the periodic case, this term would be omitted. The new definition of the strip width s'' contains two important case distinctions. The second case denotes the reduction in the size of the last dimension. Note that the intuitive assumption of $\frac{s'(i)}{2}$ does not hold, since the distribution of nodes between the two occupied strips does not have to be balanced.

The newly calculated worst-case optimality ratios can be seen in Tables 6.2 and 6.3 for the non-periodic and the periodic case respectively. Note that for the non-periodic case, the ratio sometimes is better than for unstretched partitions. The reason for that is that while for the squarest strips' hyperrectangle's surface area one edge is subtracted, the optimal hypercube surface area does not account for that. Interestingly, we can observe that for one case where $p = 5$ and $n = 2$ the worst-case hyperrectangle has even fewer outgoing edges than the perfect hypercube when it is not subjected to periodic communication across the stretched edge. Note that this is a realistic assumption, since we may argue that the grid has to be big enough that at least one partition is not able

Table 6.3: The numerically evaluated worst-case scenarios for the squarest strips algorithm, when a partition might stretch across a second strip. The comparison here is between the *periodic* case and the perfect hypercube.

p	2 dims	3 dims	4 dims	5 dims	6 dims
2	1.4142	1.6799	1.7838	1.8379	1.8708
3	1.7321	2.2435	2.4128	2.4915	2.5353
4	1.9167	1.8961	2.1508	2.2652	2.3274
5	1.6398	1.4440	1.5452	1.5821	1.5983
10	1.8974	2.0826	1.9413	2.0815	2.1446
12	1.8475	2.0986	2.0551	2.2739	2.3747
14	1.8174	1.6068	1.4277	1.4853	1.5032
16	2.0714	1.6799	1.8750	1.7121	1.7638
18	2.0203	1.7472	2.0598	1.9014	1.9786
20	1.9805	1.8096	2.2205	2.0633	2.1601
25	2.1556	1.7414	2.5491	2.3859	2.5156
30	2.0692	1.9334	2.1258	2.0138	2.1447
32	2.0428	1.9666	2.1492	2.3708	2.2180

to touch an edge. In this case, the correct worst-case ratio is the maximum value from Tables 6.1 and 6.2.

Another observation, which is especially visible in Table 6.3, is that the optimality is no longer smaller than what we have proven for the *k*-*d*-tree algorithm. When we look at specific partition sizes, for example, $p = 3$, the bound seems to increase with higher dimensions. In practice, however, a small partition of size 3 will, in the worst-case, always be arranged in a 1-wide, 3-long strip, regardless of the number of dimensions. The increasing ratio can be explained with the optimistic estimation of the optimal arrangement since the perfect hypercube's surface area is not restricted to integer divisibility.

To summarize the findings, we show a recursive procedure for calculating the worst-case ratio, which can be used to numerically approximate an upper bound for the quality of any solution. The theoretical bound of $O(2^n)$ is unfortunately not tight enough for realistically estimating the quality of mappings. The numerical evidence suggests that a much more realistic, tighter bound would be $O(3)$, however, we could not find a thorough proof for that bound yet.

CHAPTER 7

Experimental Evaluation

In order to evaluate the presented algorithms, we implemented all approaches in MPI [19], theoretically analyzed the quality of the resulting mappings and measured the communication times with a custom microbenchmark. Similarly to Gropp's [7] communication measurements, we will benchmark the runtime with non-blocking `MPI_Isend` and `MPI_Irecv` calls.

7.1 Setup

In this section, we describe the hard- and software setup that was used to conduct the benchmarks as well as how we chose to implement our testing framework.

7.1.1 Cluster Setup

All measurements were taken on our cluster Hydra, which has 36 Intel Skylake OmniPath computation nodes, each containing two Intel Xeon Gold 6130 CPUs. Each node, therefore, has 32 cores, running at 2.1GHz and 94GB of memory. On the cluster, we used Open MPI 4.0.1 and compiled with gcc version 8.3.0 and `O3` optimization flags. We relied on Slurm's default process mapping for numbering cores within the same node consecutively[1], in order to speed up the benchmarking process.

[1]To be precise, we configured our first distribution method with `block`, so that consecutive tasks share a node. The second parameter is set to `cyclic` so that both processors within a node are utilized equally. The tasks are distributed in a round-robin fashion between the two sockets. For more details, see the possible configurations for `SBATCH_DISTRIBUTION` on `https://slurm.schedmd.com/sbatch.html`.

7.1.2 Test Suites

In order to isolate the impact of specific factors, we defined our test-suites in a structured manner. We looked at both the theoretical problem definition from Section 3.1 and actual application parameters of our reference implementation and identified the defining test-variables. For each test-suite, we choose one of these test-variable and fixed all other parameters as best as possible. The identified test-variables were:

- Number of partitions
- Partition sizes
- Number of dimensions
- Dimension sizes
- Stencil size
- Stencil skew/shape
- Message size

The basic a priori hypothesis for each test-suite was the same. We expected Gropp's algorithm to perform well for cases where prime factorizations yield well-behaved factors, followed by the squarest strips algorithm due to the compact, consecutive assignments, followed by the two different k-d-tree implementations. Obviously, we also aspired to outperform MPI's standard behaviour, as well as grids with a randomly shuffled assignment. We will especially look into deviations from these expectations in the following sections for the specific test-suites.

Note that many measurements will be missing for Gropp's algorithm since we also conducted tests for which his algorithm is not eligible. For example, when fixing the grid size to be 15×15 and iterating over the number of partitions p, then it might happen that not all partitions have the same size if p does not divide $15 \cdot 15 = 225$. In these cases, we simply omitted the measurements for Gropp's algorithm and focused on the other ones.

Aside from the test-suite, which explicitly tests different message sizes, we ran all instances with message sizes of 1KB and 1MB respectively. We choose the comparatively large size of 1MB in order to get statistically significant results, without having to measure a high number of iterations, reducing the overall benchmarking time. The tests with 1KB were chosen since we also wanted to get some results for smaller instances. An additional benefit of testing two different sizes is the potential difference in favouring bandwidth (1MB) versus latency (1KB). A priori we predict that our metric, which favours communication over shared memory, is beneficial for reducing both bandwidth and latency. This means that a mapping, which performs well for 1MB, should also perform similarly well for 1KB when compared to the other solutions.

7.1.3 Implementation Details

Before any Cartesian communicators are instantiated, the configuration for a specific test run is loaded and distributed to all processes. Listing 7.1 shows the exposed configuration properties that are available to all processes. Note that we also conducted tests where not all computation nodes required the same number of processes. After distributing the configuration, each process, therefore, decided whether it is actually a part of the test or not. If not, then both `CONFIG_assigned_partition` and `CONFIG_world_comm` are set to `MPI_UNDEFINED`.

For example, let us suppose that we want to conduct a test with one partition containing only 5 processes and one partition of 10 processes. We would then start the job with two computation nodes, both having at least 10 processes assigned. The first node will then correspond to the first partition and will only assign 5 processes to `assigned_partition` 0 - the other 5 processes will have an undefined partition. The second node will assign all 10 processes and set their `assigned_partition` to 1. The new `CONFIG_world_comm` will only consist of the 15 assigned nodes. This newly created communicator will be used by all algorithms as the base communicator.

Listing 7.1: All implemented approaches could access these exposed configuration variables.

```
extern int  CONFIG_num_dims;
extern int* CONFIG_dims;
extern int* CONFIG_periods;
extern int  CONFIG_stencil_size;
extern int* CONFIG_stencil;
extern int  CONFIG_num_parts;
extern int* CONFIG_part_sizes;

// these two flags are often used in conjunction
extern bool CONFIG_use_real_memory_nodes;
extern bool CONFIG_only_analysis;

extern int CONFIG_num_communication_rounds;
extern int CONFIG_bytes_per_round_neighbour;
extern int CONFIG_bench_iters;

// this value is used for counting inter-node communication
// rather than MPI's shared memory flag
extern int CONFIG_assigned_partition;
// a modified copy of MPI's world communicator
extern MPI_Comm CONFIG_world_comm;
```

Should the test-suite call for partition sizes greater than our physical node size, then the `CONFIG_use_real_memory_nodes` flag is used and set to false. Processes from multiple, physical nodes may then be used to be assigned to a common `assigned_partition`. While this allows us to produce virtual mappings that surpass our hardware limitations, it does not make much sense to actually benchmark the communicator's performance as it does not represent our actual hardware anymore.

We then implemented a wrapper function with the same signature for all our algorithms, which constructs the Cartesian communicators. Since not all algorithms benefit from all parameters in the configuration, the wrapper is responsible for feeding the correct values to the actual implementation and storing the produced communicator. Our benchmarking suite then simply takes a list of these wrappers and measures the communication times for each of them. After all measurements have been taken, the raw, non-cumulated values, as well as every rank's position within the grid, are exported to a file in JSON format for each approach.

We will compare the mappings of all algorithms to the mappings that are produced, when calling `MPI_Cart_create`, with the reordering flag set to true. As Gropp states in his work, we could also not observe any change in the mappings whenever this reordering flag was set to true or false. In both cases, MPI's default coordinate assignment is a row-major assignment over the dimensions in their given order. Note that MPI, therefore, benefits from our assumption that processes on the same physical node are consecutively numbered. In future, we will refer to this row-major assignment as MPI's standard mapping algorithm or simply the standard algorithm.

As already mentioned, we benchmarked our communication durations similarly to Gropp, using the non-blocking `MPI_Isend` and `MPI_Irecv` calls. The required amount of data is defined in the configuration in bytes per neighbour, so we used the MPI datatype `MPI_BYTE` for the communication and `char` for the allocation in C. Listing 7.2 shows the code snippet that was used for benchmarking. Before each iteration, `MPI_Barrier` is performed and after each measurement (aka after each call of this function) the send and receive buffers are verified for correctness. This verification process is not included in the time measurement, but serves the purpose of testing our implementations for correctness (non-exhaustively) and prevents the compiler from potentially optimizing out any writes to the receive buffer. For each iteration, we then call `MPI_Allreduce` to find the maximum duration of the communication procedure over all participating processes. Note that when we refer to *average* or *minimum* runtimes in the future, we refer to the *average* or *minimum* with regard to the measured maximum durations (ignoring the first five warmup runs).

Listing 7.2: The communication procedure that was used for benchmarking. The backward- and forward neighbours array was computed prior to the runtime measurement.

```
void communicate(MPI_Comm comm, int elems, const char *send_buffer,
         char *recv_buffer, const int *forward_neighbours,
         const int *backward_neighbours) {
  MPI_Request requests[CONFIG_stencil_size * 2];
  for (int i = 0; i < CONFIG_num_communication_rounds; ++i) {
    for (int j = 0; j < CONFIG_stencil_size; ++j) {
      MPI_Irecv(&(recv_buffer[elems * j]), elems, MPI_BYTE,
                backward_neighbours[j], 1, comm, &(requests[j * 2]));
    }
    for (int j = 0; j < CONFIG_stencil_size; ++j) {
      MPI_Isend(&(send_buffer[elems * j]), elems, MPI_BYTE,
                forward_neighbours[j], 1, comm, &(requests[j * 2 + 1]));
    }
    MPI_Waitall(CONFIG_stencil_size * 2, requests, MPI_STATUSES_IGNORE);
  }
}
```

7.2 Experimental Results

We present the results for each test-suite separately and discuss the observations accordingly. If not stated otherwise, each measurement was performed with 1MB per neighbour with 100 iterations (discarding the first few). For a few particularly striking runs, we will also take a deeper look into the specific mappings and the statistical significance of the measurements. Since we performed hundreds of runs in total, we only discuss a selection of tests. Graphs for the complete test-suites can be found in the appendix. In Section 7.3 we discuss the algorithms' behaviour for smaller message sizes, as the observations yield surprising results.

7.2.1 Partitions

For the first suite we choose to fix the dimensions (number and size) as well as the stencil, but vary the partition number $|P|$ and sizes to fit within the fixed constraints. We performed this benchmark on a 15×15 (2D), a 20×10 (2D) and a $6 \times 6 \times 6$ (3D) grid with a simple 5pt-stencil. We let $|P|$ go from 2 to 30 in steps of 2 and then calculated each partition's size so that the specified grid could be filled. To fill the grid, some partitions are 1 bigger than others. Note that for some measurements Gropp's algorithm, therefore, is not able to run, due to its partition size constraint. Figures 7.1 show the absolute runtimes of the measured algorithms on a 15×15 grid, while Figure 7.2 shows both optimality metrics. We can see that, in general, the differences in both metrics correlate with the runtime differences for all approaches. Interestingly, the k-d-tree algorithms often outperformed the squarest strips algorithm, even though they were a little worse wrt. the optimality criteria. Gropp's approach was not compatible with any tested partition configuration on a 15×15 grid. For the other approaches, it can be seen that

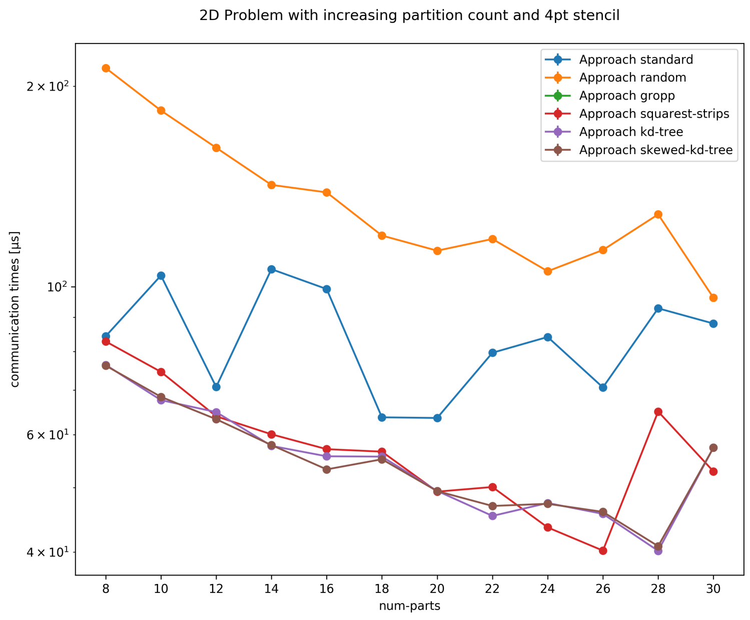


Figure 7.1: The minima of all measured maximum runtimes for a 15×15 grid, 5pt-stencil and 1MB of data per neighbour. The grid was not changed, but the partition sizes were (uniformly) adapted according to the varying number of partitions. Unfortunately, Gropp's algorithm was not compatible with any of these configurations and therefore cannot be seen here.

they indeed perform as expected. The random process assignment, as well as MPI's standard behaviour, did not yield good mappings, while the squarest strips algorithm is very comparable to the two k-d-tree approach for the most part. The only two significant exceptions are the speed-up of the squarest strips algorithm for 26 and its noticeable shortcoming for 28 partitions.

For the case with 26 partitions, we can explain this behaviour, since the grid is split into 17 partitions of size 9 and 9 partitions of size 8. The squarest strips algorithm succeeds in mapping all 17 9-big partitions into perfect 3×3 squares while still maintaining an acceptable shape for the 8-sized partitions. The squarest-strips' worse performance for 28 partitions can be explained similarly. Since there is only one partition of size 9 and 27 partitions of size 8, the algorithm wrongly tunes the strip-size to the biggest partition of size 9. Had it tuned for size 8, then the target bounding rectangle would be 2×4, which would probably have been preferable in this case. The mappings for all algorithms for these instances can be seen in Figures 7.3 and 7.4 respectively.

The results for the 20×10 grid showed a less obvious trend, which can be seen in Figure 7.5. This time, Gropp's algorithm was able to find solutions for multiple runs. Generally speaking, the same observations still hold true, but for 20 and 26 partitions, the

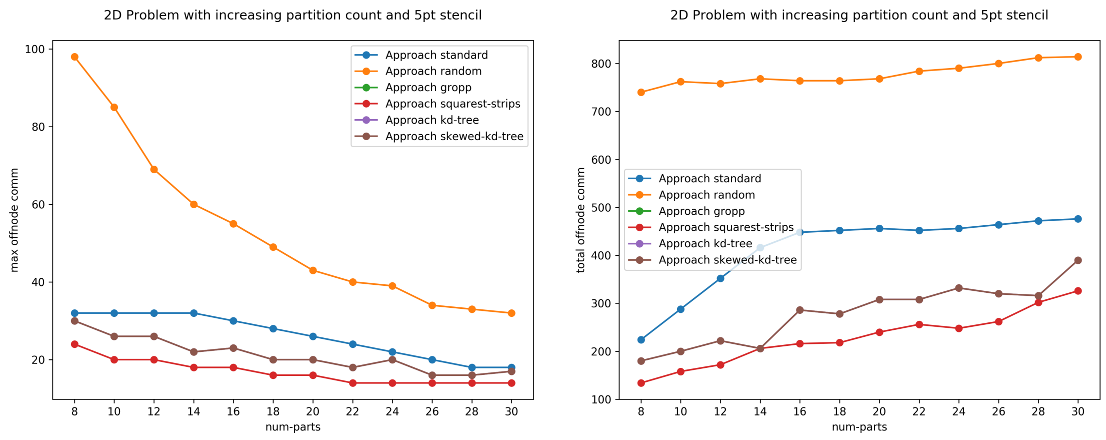


Figure 7.2: The two graphs show the maximum (left) and total (right) off-node communication edges for the test setup for a 15 × 15 2D grid with an increasing number of partitions. For the most part, the previously shown differences in runtime correlate with both the total and maximum number of outgoing edges, however, it can be seen that even though the squarest-strips algorithm had the best optimality metrics in both cases, it was often outperformed by the *k*-*d*-tree algorithm in terms of runtime. Note that both *k*-*d*-tree approaches had the exact same mappings since the stencil was not skewed.

standard algorithm performed better than other approaches. We can see the mapping for the more pronounced case with 20 partitions in Figure 7.7. Since the grid was arranged favourably for the standard algorithm, we can see that it assigned each 10 × 1 partition to its own row. The squarest strips, as well as the *k*-*d*-tree algorithms, did not manage to find rectangular partitions but managed to find quite compact mappings nevertheless. Gropp's algorithm yielded the intuitively best result of arranging the partitions in 2 × 5 rectangles, however, performed worse than the other approaches. With only 14 maximum outgoing edges for the bottleneck partition, Gropp's and the squarest strips algorithm's mapping are not only intuitively good, but *better* than the standard algorithm's mapping, which has 20 outgoing edges from its bottleneck partition. Note that our metrics did not predict this behaviour, as can be seen in Figure 7.6, where the standard algorithm performs unusually well, but never outperforms the squarest strips approach. To show that this is no accident, Figure 7.8 shows boxplots for all approaches' measurements. Interestingly, while Gropp's mapping did not yield the best result, its consistent nature appears to result in a very small variance in measurements. We discuss this paradox observation further in later sections.

Secondly, we fixed the number of partitions and dimensions but varied the dimension and partition sizes. In other words, we let the grid grow (in the shape of a hyper-square) proportionally to the growth of the partitions. We did this for 5, 10 and 30 partitions and for 2 and 3 dimensions. We let the dimensions grow from 4 to 16 in steps of two and calculate the partition sizes in the same manner as before. The main difference, in

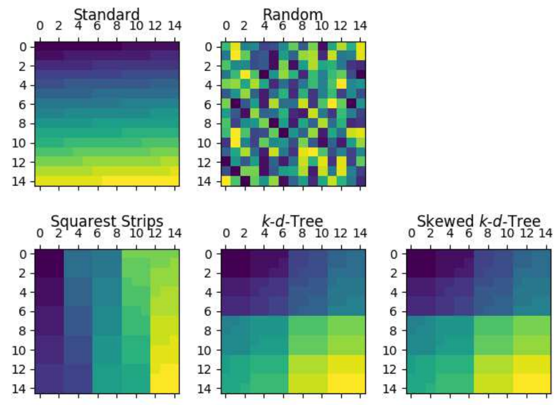


Figure 7.3: The different algorithms' mappings on a 15 × 15 grid with 17 partitions of size 9 and 9 partitions of size 8. It can be seen that the squarest strips algorithm manages to fit all 9-sized partitions into 3×3 squares, while still being able to accommodate the 8-sized partitions appropriately. While the *k*-*d*-tree approaches yield overall quite compact results, it can be seen that multiple partitions are split into two disjoint components.

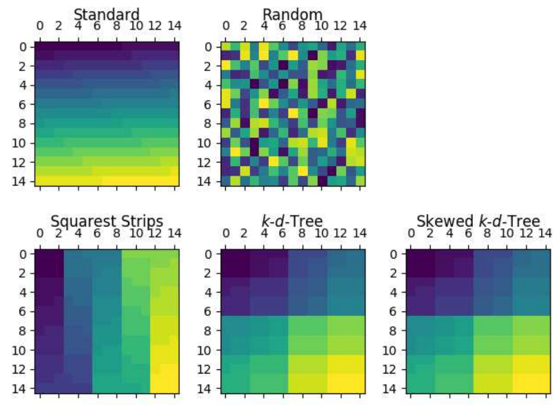


Figure 7.4: The different algorithms' mappings on a 15 × 15 grid with one partition of size 9 and 27 partitions of size 8. Since the squarest strips algorithm adapts its strip size to the biggest partition, it therefore does not account for the fact, that the majority of the participating partitions are actually smaller (8 instead of 9). The *k*-*d*-tree approach, on the other hand, appears to make the cuts along edges, which mostly result in two disjoint, but individually square components.

comparison to the first test-suite, is that we do not change the parameters in a way where inherently different mappings are expected between runs. Since partitions and dimensions are scaled up at the same rate, we expect an algorithm that performs well for one run, to perform similarly for another one. Figure 7.9 shows the measurements for the instance with 10 partitions on a 2D grid. As expected, all approaches show a clear trend and the runtime scales with the number of processes. Interestingly, the 10 × 10-grid instance shows very similar runtimes for the standard, squarest strips and *k*-*d*-tree algorithms, but a good performance for Gropp. Looking at the mapping for this 10 × 10 instance in Figure 7.10, we can see Gropp's 2 × 5 rectangles again, this time performing better than the standard algorithms 1-wide partitions.

Lastly, we fixed the partition sizes and the number of dimensions and varied the dimension sizes and partition number. This is very similar to the second suite, but we are not changing the *size* of the partitions, but the *number* of partitions instead. Even though we assume a fixed partition size, it may still vary by 1, since the grid has to be filled exactly. Figure 7.11 shows the results for a 2D grid and a fixed partition size of 10. Here, the two *k*-*d*-tree approaches perform consistently well, only rarely being outperformed by Gropp or the squarest strips algorithm.

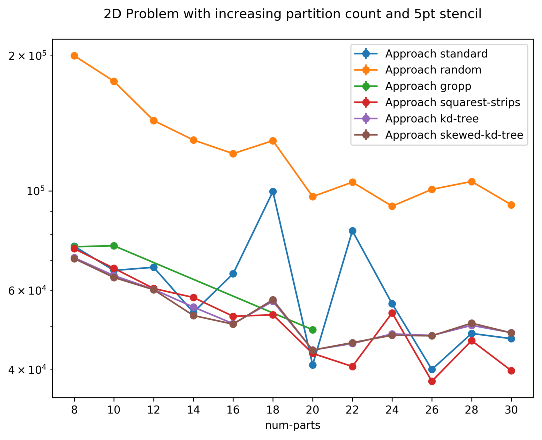


Figure 7.5: The minima of all measured maximum runtimes for a 20 × 10 grid, 5pt-stencil and 1MB of data per neighbour. The grid was not changed, but the partition sizes were (uniformly) adapted according to the varying number of partitions. Interestingly, the standard algorithm found very well-performing mappings for multiple instances.

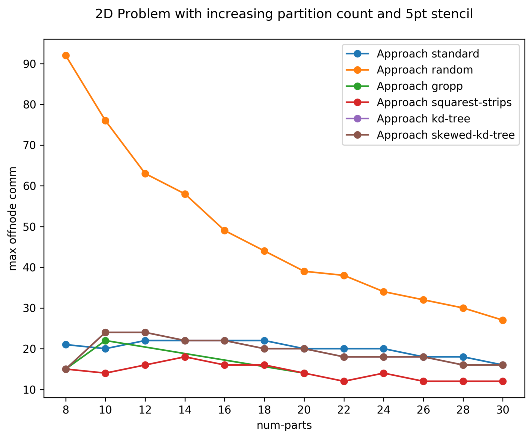


Figure 7.6: The maximum number of offnode communication edges for the instance with a 20×10 grid and 5pt-stencil. It can be seen that the standard algorithm performs similarly to the *k*-*d*-tree approaches.

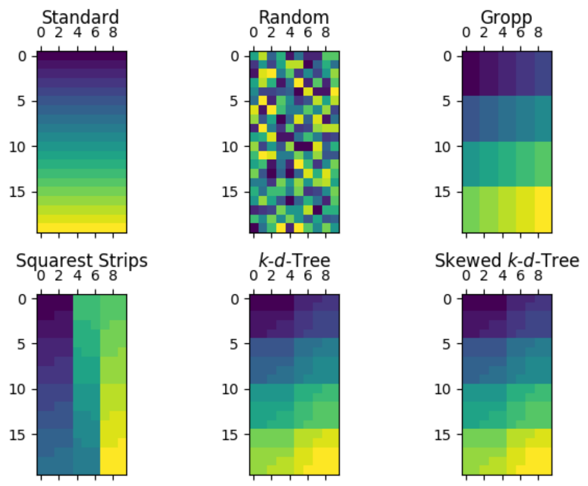


Figure 7.7: The different algorithms' mappings on a 20×10 grid with 20 partitions of size 10. Interestingly, the standard behaviour's row-major mapping yielded the best runtime, while Gropp's seemingly compact solution performed not as good as the bottom three approaches.

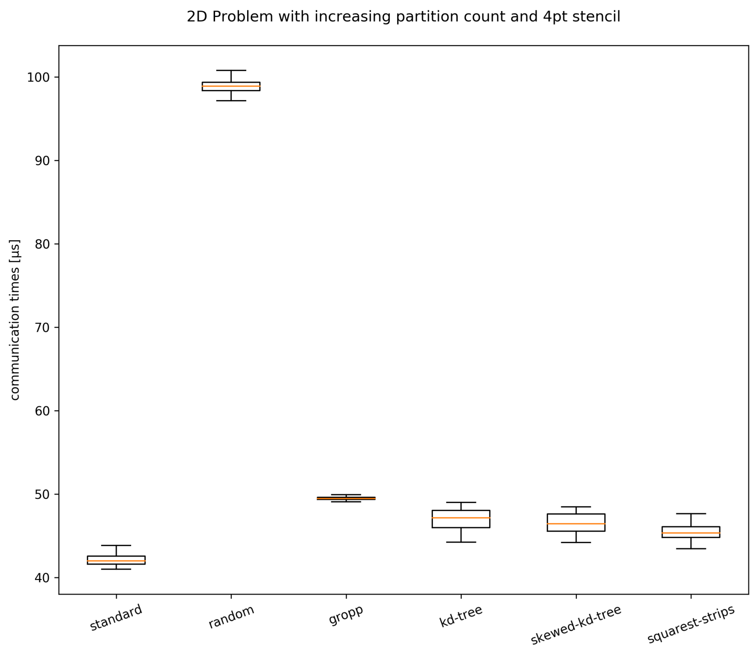


Figure 7.8: The different algorithms' runtimes on a 20×10 grid with 20 partitions of size 10. It can be seen that the standard algorithm performed best, followed by the squarest-strips and k-d-tree algorithms. Gropp's algorithm, while not performing as well as the others, yielded the most consistent runtime measurements.

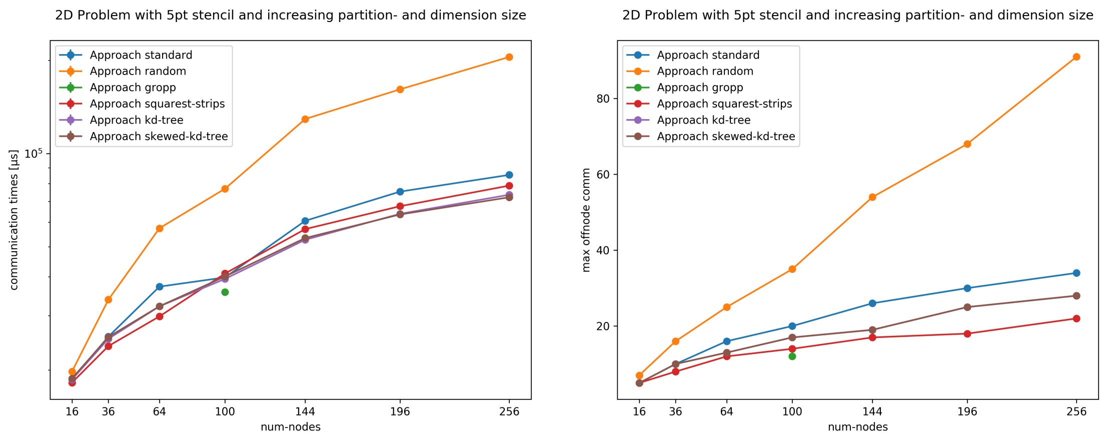


Figure 7.9: The minima of all measured maximum runtimes (left) and the maximum number of offnode communications (right) for a square-like grid, 10 partitions, 5pt-stencil and 1MB of data per neighbour. It can be seen that all algorithms' runtimes scale as expected with increasing partition sizes. Gropp's approach only worked for the 10×10 grid, where he outperformed all others.

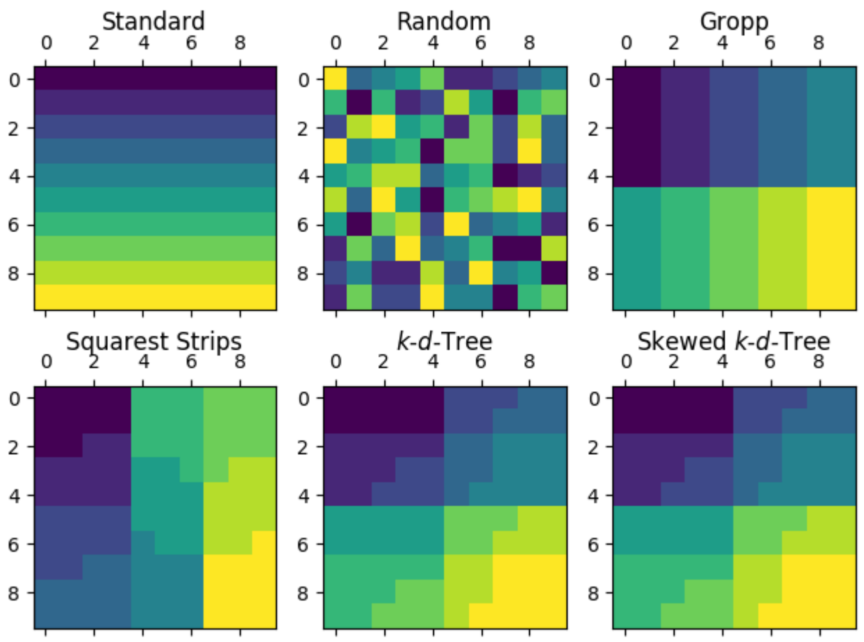


Figure 7.10: The different algorithms' mappings on a 10 × 10 grid with 10 partitions of size 10.

7.2.2 Dimensions

In the previous section, we already varied over some aspects of the grid's dimensions. However, these changes were always made in reaction to modifications to the partitions. Here we will focus on changes with respect to the dimensions. We start by fixing the partitions and stencil and vary the number of dimensions while still accommodating the same partitions as best as possible. If necessary, partition sizes will be off by 1 again. We performed these tests with 10 or 30 partitions, each having sizes of 10 and 30. We increased the number of dimensions and calculated the desired grid size.

Figure 7.12 shows that - contrary to previous test-suites - the squarest strips algorithm seldom performs best. It was to be expected that the *k*-*d*-tree algorithm performs good for higher dimensions since we proved an increasingly better optimality ratio for higher dimensions in Section 5.3.3. Unfortunately, we currently cannot explain why the random communicator performed so well for 5 dimensions, leaving the squarest strips algorithm behind, since our optimality metrics, which can be seen in Figure 7.13, always predict the standard and random assignments to perform worse.

The other suite fixed the partitions and stencil and varied the dimensions' sizes. We ran these tests with 10 or 30 partitions, each having sizes of 10 and 30 on grids of 2 and 3 dimensions. Since the grid's size is already constrained by the partitions and number of dimensions, this test-suite tests different dimension *skews*. The x-axis in Figure 7.14

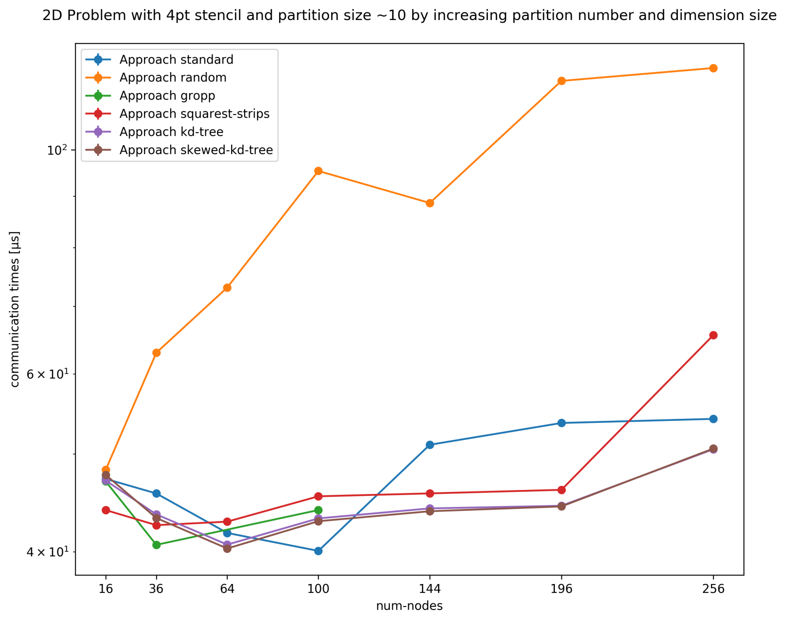


Figure 7.11: The minima of all measured maximum runtimes for a square-like grid, varying partitions of size 10, 5pt-stencil and 1MB of data per neighbour. Again, the run for the 10×10 grid is an outlier. For this one run, the standard approach outperforms the other algorithms.

therefore shows the ratio between the first and the second dimension.

Looking at the optimality comparison in Figure 7.15, we could predict that the squarest strips algorithm consistently performs best, followed by either the *k*-*d* or the standard algorithm. However, for most runs in Figure 7.14, we can see that both *k*-*d*-tree variants perform best most of the time, followed by the squarest strips and standard algorithm. Interestingly, we can see that for skews of 2.9, 3 and 3.1, the standard algorithm performs well, especially compared to Gropp's approach. For the case with 3.1, the reason most probably is the 10x31 grid with 10 partitions of size 31[2]. Since 31 is a prime number, Gropp's algorithm has no other choice, but to arrange the partitions into 1-wide strips, while the standard algorithm benefits from the comparatively narrow first dimension. The row-major assignment, therefore, results in a compact bounding rectangle of roughly $10 \times (4 \pm 1)$. In the instance for a skew of 3, the mappings were created for a 10×30 grid. The standard behaviour's row-major assignment managed to create perfect 10×3 rectangular partitions, while Gropp's algorithm chose to go with 2×15 rectangles instead, hindering the performance. Interestingly, the squarest strips algorithm managed to find 5×6 partitions - the best possible solution wrt. the bottle-neck value - but still managed

[2]We originally generated this suite with partitions of size 30 in mind. A test run where all partitions have size 31 is the result of fitting 10 partitions of size 30 into a 10×31 grid *as best as possible*.

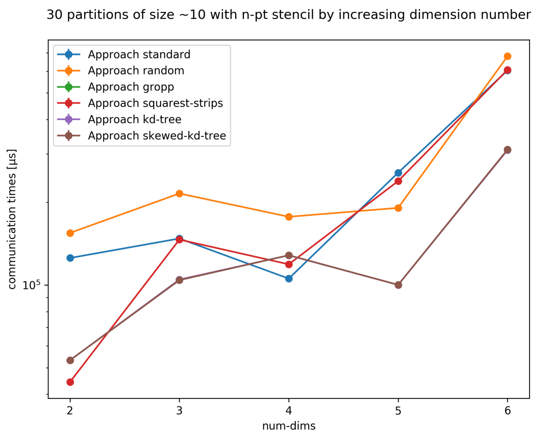


Figure 7.12: The minima of all measured maximum runtimes for 30 partitions, roughly of size 10 and 1MB of data per neighbour.

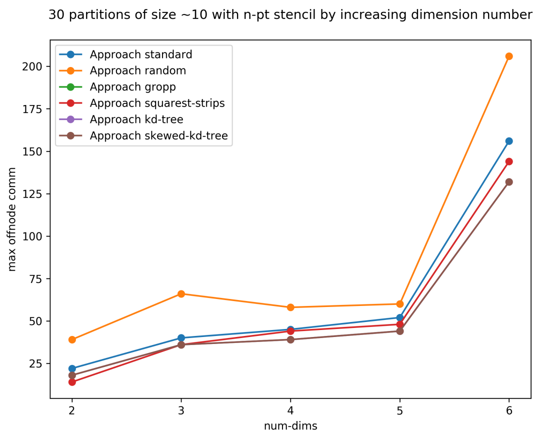


Figure 7.13: The maximum number of offnode communications for 30 partitions, roughly of size 10. It can be seen that while the differences are small, the standard and random approaches are always outperformed by the squarest-strips and *k*-*d*-tree approaches.

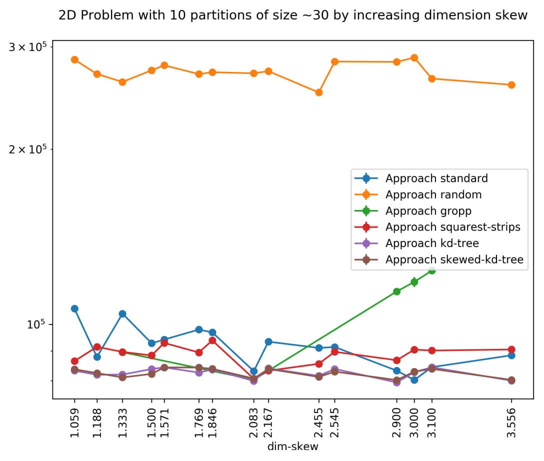


Figure 7.14: The minima of all maximum communication times for 10 partitions of roughly size 30 on a 2D grid. It can be seen that, most of the time, the *k*-*d*-tree approaches perform best. For the runs with skew 2.9, 3 and 3.1 the standard algorithm performed better than usual, while Gropp's approach falls behind in these cases.

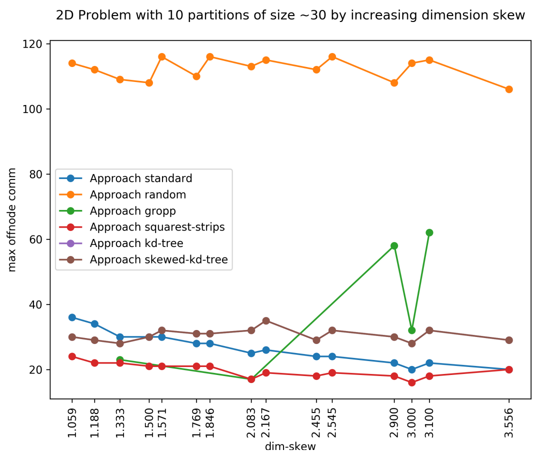


Figure 7.15: The maximum number of offnode communications for 10 partitions of roughly size 30 on a 2D grid. It can be seen that, the squarest strips algorithm consistently performs best. the standard algorithm managed to get better mappings than the *k*-*d*-tree approaches for some instances with a comparatively high dimension skew.

to perform worse than the standard algorithm in terms of communication time. These mappings can be seen in Figure 7.16.

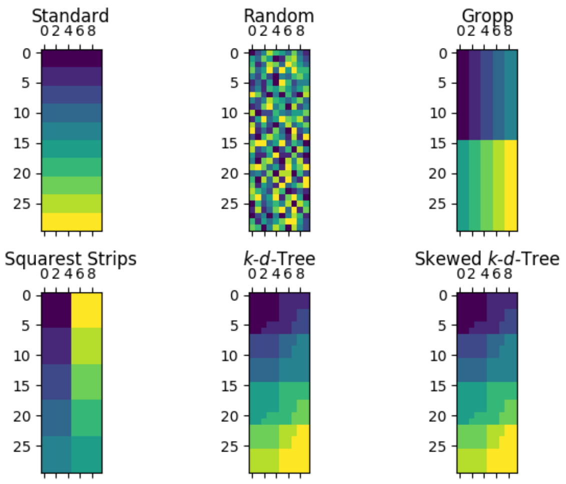


Figure 7.16: The different algorithms' mappings on a 10×30 grid with 10 partitions of size 30.

7.2.3 Stencils

In this section, we benchmarked the effect of different stencils on grids that were already tested in previous suites. Firstly, we fixed both the partitions and dimensions and increased the stencil's size uniformly in all directions. The resulting stencils, therefore, should be unskewed and all algorithms are expected to perform according to previous results. Listing 7.3 shows the Python code, which was used to generate these stencils. The `index` parameter indicates the recursion depth and therefore how many vectors are added to the stencil.

Listing 7.3: The helper function that was used for generating increasingly big stencils, while maintaining skew. Written in Python 3.

```python
def get_symmetric_stencil(index, num_dims):
  if index < 0:
    return []
  stencil = [0] * (num_dims * 2)
  dim = index % num_dims
  val = (index // num_dims) + 1
  stencil[dim] = val
  stencil[dim + num_dims] = -val
  return get_symmetric_stencil(index - 1, num_dims) + stencil
```

As already mentioned, the stencil grows uniformly in all directions, except for odd

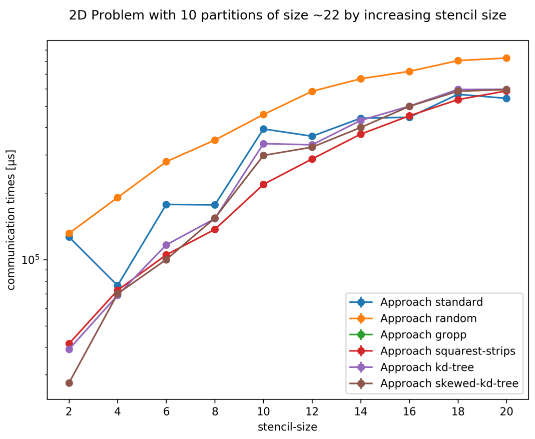


Figure 7.17: The minima of all measured maximum runtimes for a 15 × 15 grid, 10 partitions of size 22 and increasing stencil size. The difference between the skewed and unskewed *k*-*d*-tree algorithms can be seen for sizes, which are not multiples of 4.

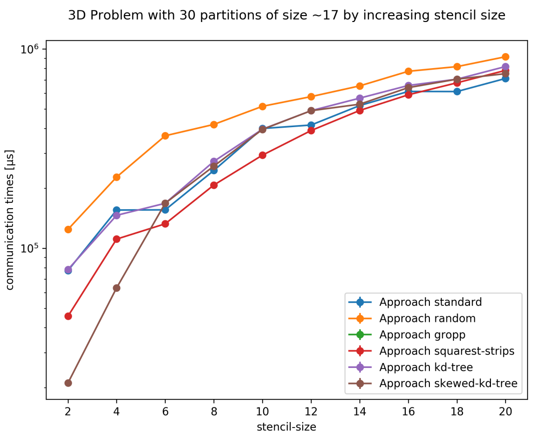


Figure 7.18: The minima of all measured maximum runtimes for a 8 × 8 × 8 grid, 30 partitions of size 17-18 and increasing stencil size. Note that the skewed-*k*-*d*-tree's runtime converges with the default *k*-*d*-tree algorithm for stencil size multiples of 6 in 3 dimensions.

indices[3]. For these cases, we expect a slight, but noticeable performance boost for the skewed-*k*-*d*-tree algorithm, especially for smaller stencil sizes. Figures 7.17 and 7.18 support these assumptions.

In Figure 7.17 we can see that the instances where the stencils' size is *not* a multiple of 4, the skewed-*k*-*d*-tree performs better than the default *k*-*d*-tree approach. The effect is most visible for a stencil size of 2, which corresponds to the simple, one-dimensional component stencil, but still pronounced for sizes of 6 and 10. Most notably, all runtimes seem to converge for larger stencil sizes, which is to be expected, since after a certain point, a partition's number of outgoing edges far surpasses the number of nodes within a partition, for any given algorithm. Figures 7.19 and 7.20 show the mappings for both of these cases.

Figure 7.18 shows similar results for the 3-dimensional case. The skewed version of the *k*-*d*-tree outperforms its non-skewed counterpart for all instances where the stencil size is not a multiple of 6. These differences aside, in both the 3D and 2D case, the squarest-strips algorithm outperformed the others for many instances.

In order to truly test the impact of a skewed stencil, the following test-suite ran different stencils on the same 2D grid and partition configuration. In order to increase the stencil's skew, we added communication edges in an unbalanced way. We started with an unskewed

[3]Odd is only the correct term for 2D-grids. More correctly, the stencil has a slight skew if the `index` + 1 does not divide the number of dimensions.

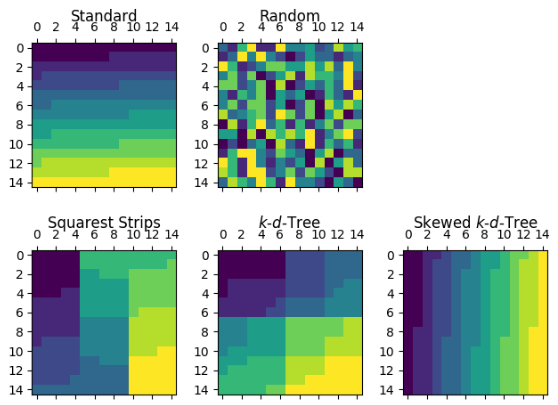


Figure 7.19: The mapping for the 2-stencil-size instance, where the two k-d-tree approaches' difference was most pronounced. Since the stencil is just a one-dimensional strip, the skewed version attempts to elongate all partitions as much as possible, by weighting the corresponding dimension infinitely higher than the other one.

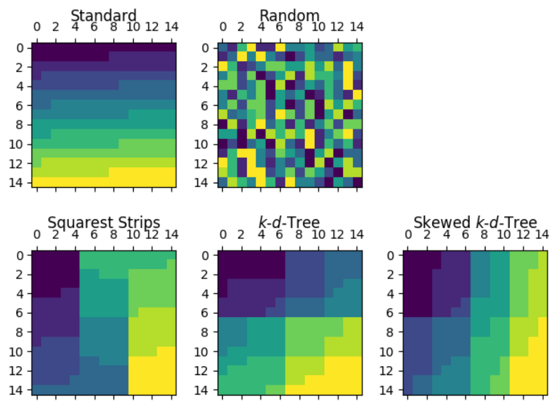


Figure 7.20: In comparison to Figure 7.19, the only difference in setup is a slightly larger stencil (size 6). The comparatively smaller skew results in slightly vertically stretched partitions for the skewed-k-d-tree approach in comparison to the default one.

5pt-stencil and progressively added more edges to the second dimension or replaced existing edges in the first dimension, with diagonals. Figure 7.21 shows the last, most skewed stencil, which was tested.

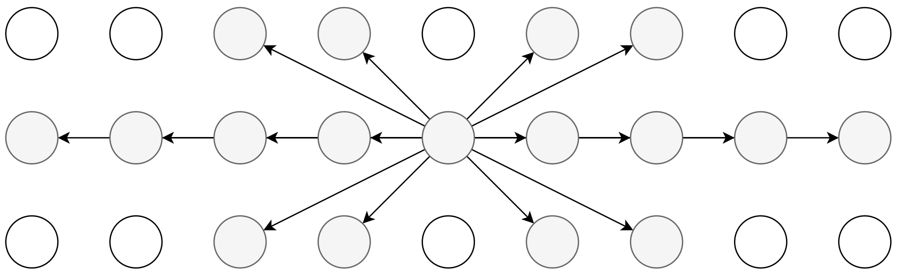


Figure 7.21: The most skewed stencil, which was used for the stencil-skew test-suite. For each step in increasing the skew, either horizontal edges were added, or vertical edges replaced by diagonal edges, starting from a 5pt-stencil.

In Figure 7.22 we can see that the most consistently best-performing algorithms are the skewed-k-d-tree, squarest strips and Gropp's approach. The latter two are interesting since they do not explicitly take the different stencils into account and still perform similarly to the skewed-k-d-tree version. As expected though, the skewed version consistently outperforms the unskewed k-d-tree variant.

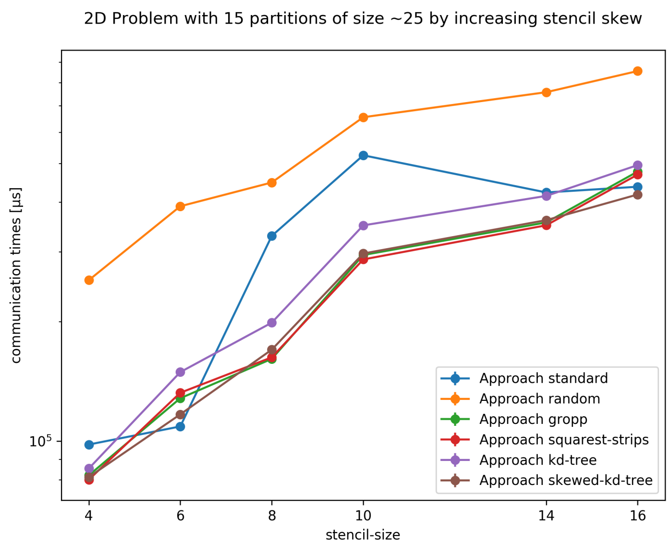


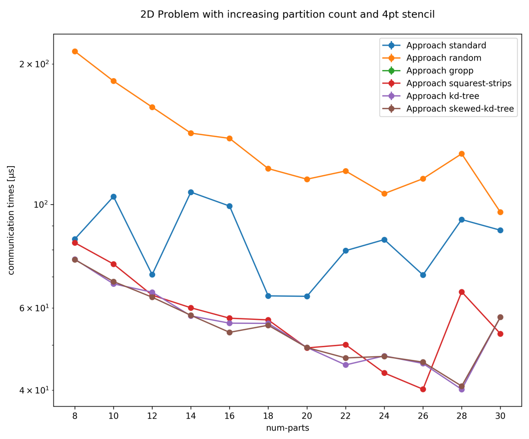
Figure 7.22: 15 partitions on a 2D grid with an increasing, non-balanced stencil-size. As expected, the skewed *k*-*d*-tree approach consistently outperforms the unskewed version here. Gropp's mappings' runtimes are very comparable to the squarest strips algorithm's. Both manage to outperform the other approaches in some instances, even though they do not explicitly take the stencil into account.

7.3 Results for smaller Message Sizes

So far, all presented results have been for comparatively large message sizes of 1MB per neighbour. For the sake of comparison, we will present some graphs that were already shown in Section 7.2, as well as their 1KB counterparts. Note that - besides the message size - every other aspect of the test configuration is the same, meaning that all mappings (except for the random communicator) will be equal for 1MB and 1KB benchmarks.

7.3.1 Comparison with existing Results

Figure 7.23 shows a comparison between the measurements for 1MB (left) and 1KB (right) on a 15×15 grid and increasing partition numbers. It becomes apparent that *good* mappings, by the definition of our optimality criteria, perform well for 1MB, but badly for the other tests. This phenomenon is not restricted to this specific test-suite but could be observed during all tests. For example, Figure 7.24 shows another comparison, where Gropp's approach is also represented. Again, the random rank assignment consistently outperforms all other approaches.



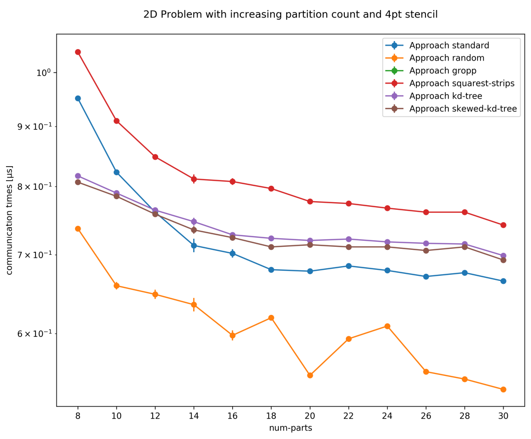
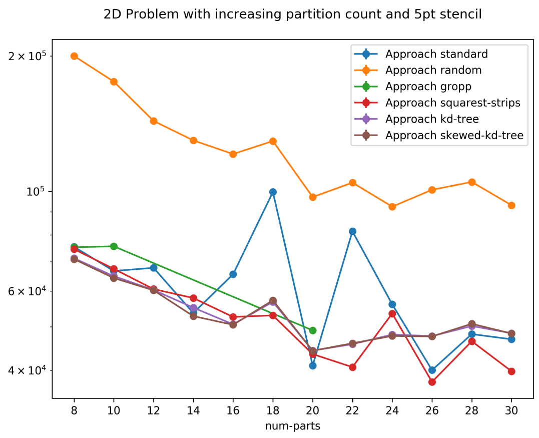


Figure 7.23: The two graphs show the same test setup for a 15 × 15 2D grid with an increasing number of partitions. The only difference is the number of bytes transferred per neighbour, which is 1MB for the left and 1KB for the right side. It can clearly be seen that the algorithms, which performed well for one side perform badly on the other.



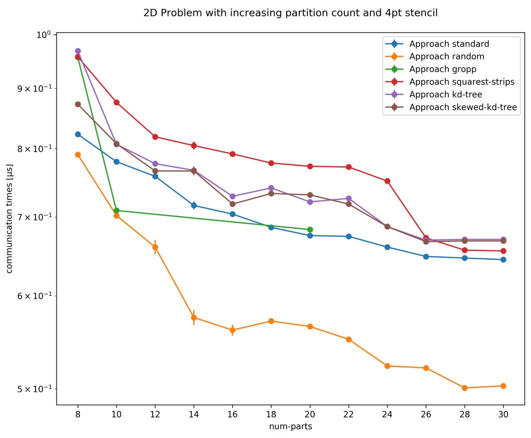


Figure 7.24: The two graphs show the same test setup for a 20 × 10 2D grid with an increasing number of partitions. As before, the only difference is the number of bytes transferred per neighbour, which is 1MB for the left and 1KB for the right side.

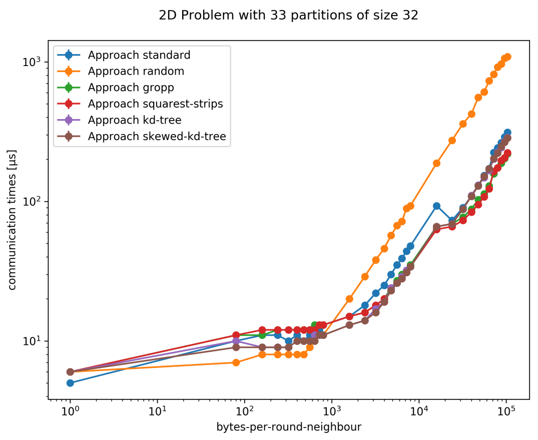


Figure 7.25: The minima of all measured maximum runtimes for a 44 × 24 grid, 33 partitions of size 32 and 5pt-stencil. Note that both axes are logarithmic in this graph.

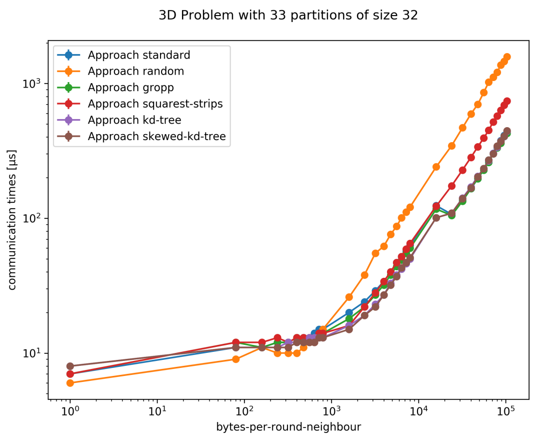


Figure 7.26: The minima of all measured maximum runtimes for a 12 × 11 × 8 grid with 33 equal-sized partitions, each having 32 processes. Note that both axes are logarithmic in this graph.

7.3.2 Varying Message Sizes

After the surprising results from the previous section, we formulated a new hypothesis before conducting another test-suite. We assumed that there must be some point, at which a mapping's predicted quality starts to correlate with the actual runtime. For that reason, we created a large test instance, which saturates all processes on our cluster with 200 repetitions in order to ensure statistical confidence and that no other test-instances could be running at the same time.

The first test was conducted on a 44 × 24 grid with 33 equal-sized partitions, each having 32 processes. We used the 5pt-stencil in order to get results where all algorithms perform as intended. Figure 7.25 shows the communication times for increasing message sizes. Note that both the x- and y-axes are logarithmic for this test. Although the graph is quite busy, it can clearly be seen that all measurements with fewer than 1000 bytes per neighbour favour the random mapping, instances between 10^3 and 10^4 bytes favour the k-d-tree algorithms and instances with even bigger message sizes perform well when using Gropp's or the squarest strips mapping.

The second test was conducted on a 12 × 11 × 8 grid with 33 equal-sized partitions, each having 32 processes. We used a 7pt-stencil - the three-dimensional pendant to the two dimensional 5pt-stencil - in order to get results where all algorithms perform as intended. Figure 7.26 shows the communication times for increasing message sizes. Again, both the x- and y-axes are logarithmic. Similarly, we can see that the random algorithm performs best for smaller message sizes and the k-d-tree algorithms after that. For this instance, however, the squarest strips algorithm does not outperform the others for message sizes beyond 10^4 bytes. This is especially surprising since the maximum offnode communication with 64 for the squarest-strips algorithm is smaller than the 71 and 80 for the k-d-tree and Gropp's approaches respectively. The cumulated number of outbound communication edges is also lower with 1522 for the squarest strips, compared to the 1928 and 2272 for k-d-tree and Gropp. A possible conclusion from these results is that the squarest-strips algorithm might have overtaken the others after increasing the message sizes even further, but we will reflect on these possibilities in the conclusion.

7.4 Communicator Construction Times

In this thesis, we focused on two performance indicators of the mapping problem, which we discussed in each algorithm's separate chapter. The communication time when using the constructed communicators, as well as the time it takes to create them. In this section, we will focus on the latter aspect. After having done a complexity analysis for each algorithm, the expected result would normally be that the squarest-strips algorithm outperforms both k-d-tree approaches, followed by Gropp's algorithm. Since Gropp's algorithm requires some inter-node communication during construction, we hypothesise his performance to be the worst. However, since the k-d-tree algorithm is beautifully compact and more memory efficient, we predict it to perform better than the squarest-strips algorithm, which contains a maze of `if/else` branches.

We measured the time it takes each process to calculate its new coordinate without the actual construction of the Cartesian communicator with `MPI_Cart_create`. The reason for leaving out this call was that it is equal for all algorithms and takes a long time, compared to the algorithmic work. As a result of this assumption, we only have measurements for both k-d-tree approaches, the squarest strips algorithm as well as Gropp. The standard algorithm and the random assignment have no prior algorithmic work to do and therefore have an additional computation time of 0.

In contrast to the experiments from Section 7.2, we have not repeated these measurements sufficiently many times for each test instance. Even though the inputs are slightly different, we cumulated the communicator construction time measurements from whole test-suites to reduce the variance. Figure 7.27 shows a boxplot over all creation times for more than 800 different test instances. We can see that all 3 of our algorithms performed quite similarly, with no statistically relevant difference to be seen for the average runtimes. We can, however, see that the squarest strips algorithm appears to perform the most consistent, as its fences are the narrowest of all. This is expected since its runtime is only directly dependent on the number of dimensions, which has stayed the same for most test-suites. Gropp's algorithm, on the other hand, suffered from the fact, that two separate communication rounds have to be conducted. We believe that the prime factorization, which is responsible for the comparatively bad algorithmic complexity, is not the culprit here, as the involved prime numbers are quite small and never exceeded the two-digit range.

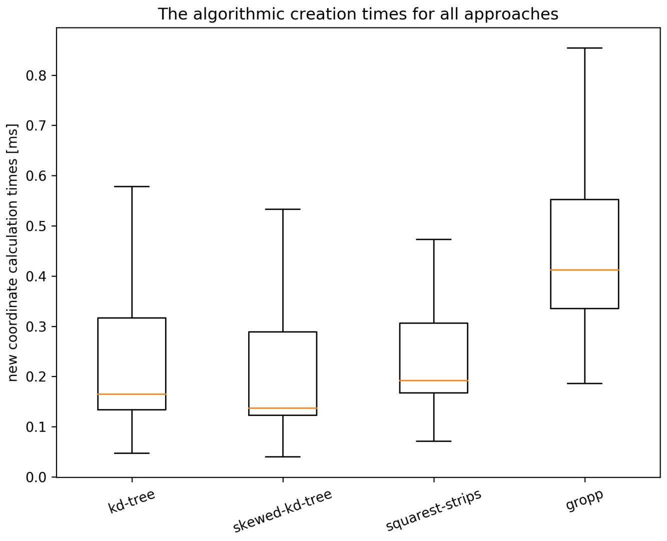


Figure 7.27: The communicator creation times for over 800 different test instances. All 3 of our algorithms outperformed Gropp's creation time on average.

CHAPTER 8 Discussion

In this chapter, we first summarize our contributions to the mapping problem and the results of our presented approaches in general. After that, we give an overview of potential future work, based on some interesting observations during the experimental phase of this thesis.

8.1 Conclusion

In this thesis, we attempted to solve the mapping problem from Cartesian grids to physical cores. After having defined our optimality criteria, we showed NP-hardness by reducing multi-way-partitioning to the mapping problem in Chapter 3. We analyzed Gropp's algorithm with its strengths and shortcomings in Chapter 4 and proved that the mappings that his algorithms produce may become arbitrarily bad for some instances. In an attempt to introduce an upper bound to the quality of possible mappings, we presented three different algorithms - the k-d-tree approaches and the squarest strips algorithm - which improve on Gropp's algorithm in terms of applicability, runtime and in some cases even quality with respect to our criteria.

Our experimental results from Chapter 7 show our improvements for many different test-suites. In addition to the expected results, we could also observe cases, where the runtimes did not directly correlate with the mappings' qualities - or at least only to a certain degree. Section 7.3 shows interesting differences in relative runtimes, just by altering the transmitted amount of data per neighbour.

8.2 Future Work

We discovered multiple aspects that should be improved in the future, starting with the MPI standard itself. The current interface for constructing Cartesian communicators is very basic. It offers no possibility for defining (weighted-) stencils or any other way for defining a preferred remapping strategy. As we discover in Section 7.3, defining the amount of data per neighbour should also impact the mappings.

Our approaches do not focus on layered hardware architectures as, for example, Niethammer and Rabenseifner [20] did. Since the *k*-*d*-tree algorithms implicitly harness consecutively numbered processes, regardless of inter-node, inter-rack, etc. properties, we somehow covered layered architectures already. We would, therefore, like to benchmark the algorithms as they are on multi-tier machines and tweak them accordingly. In order to be able to do so, the current MPI standard should offer more possibilities of identifying a process' hardware properties. Currently, the only working, standardized distinction is `MPI_COMM_TYPE_SHARED`, which only differentiates between intra- and inter-node communication. The OpenMPI implementation offers 12 more flags, such as `OMPI_COMM_TYPE_L3CACHE`, `-_SOCKET` or `-_CLUSTER`, but we could not get any of them to work[1].

Another thing that should be explored in future, is the assumption that intra-node communication is good and fast and inter-node communication is slow. While this black and white assumption seems to hold for larger message sizes of 1MB, there were multiple exceptions to it. For smaller message sizes, this assumption clearly does not to hold anymore. Also note that while messages of 1KB may be small for benchmarks, these message sizes are not unreasonable for some applications. We recommend surveying the impact of different message sizes on different systems and defining a more fitting model for describing a mapping's quality.

Due to the broad, unrestricted applicability, good runtime-scaling and the lightweight implementation of our presented approaches, they could be used by MPI implementations, if the remapping flag is set, since they outperformed the current standard behaviour in almost every instance. Both *k*-*d*-tree approaches should work out of the box since they require no additional information about the hardware or any special setup.

[1]These flags have existed since OpenMPI 3.0. The interested reader can try out these flags themselves. A full list can be found on the man page for `MPI_Comm_split_type` under `https://www.open-mpi.org/doc/v3.0/man3/MPI_Comm_split_type.3.php#toc8`.

List of Figures

List of Tables

Bibliography

[1] Prakash K. Aithal, U. Dinesh Acharya, and Rajesh G. MPI based edge detection of coloured image using laplacian of gaussian filter. *IJCA Proceedings on International Conference on Information and Communication Technologies*, ICICT(2):5–7, 2014.

[2] S. H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, 30 (3):207–214, 1981. ISSN 0018-9340. doi: 10.1109/TC.1981.1675756. URL `https://doi.org/10.1109/TC.1981.1675756`.

[3] Rainer E. Burkard, Stefan E. Karisch, and Franz Rendl. Qaplib – a quadratic assignment problem library. *Journal of Global Optimization*, 10(4):391–403, Jun 1997. ISSN 1573-2916. doi: 10.1023/A:1008293323270. URL `https://doi.org/10.1023/A:1008293323270`.

[4] E.K. Burke, R.S.R. Hellier, G. Kendall, and G. Whitwell. Complete and robust no-fit polygon generation for the irregular stock cutting problem. *European Journal of Operational Research*, 179(1):27 – 49, 2007. ISSN 0377-2217. doi: https://doi.org/10.1016/j.ejor.2006.03.011. URL `http://www.sciencedirect.com/science/article/pii/S0377221706001639`.

[5] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979. ISBN 0716710447.

[6] Timothy Gowers, June Barrow-Green, and Imre Leader. *The Princeton companion to mathematics*. Princeton University Press, 2008.

[7] William D. Gropp. Using node information to implement MPI cartesian topologies. In *Proceedings of the 25th European MPI Users' Group Meeting*, EuroMPI, pages 18:1–18:9, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-6492-8. doi: 10.1145/3236367.3236377. URL `http://doi.acm.org/10.1145/3236367.3236377`.

[8] Torsten Hoefler and Marc Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proceedings of the International Conference on Supercomputing*, ICS, pages 75–84, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0102-2. doi: 10.1145/1995896.1995909. URL `http://doi.acm.org/10.1145/1995896.1995909`.

[9] Shu-Ming Hsieh, Chiun-Chieh Hsu, and Li-Fu Hsu. Efficient method to perform isomorphism testing of labeled graphs. In Marina L. Gavrilova, Osvaldo Gervasi, Vipin Kumar, C. J. Kenneth Tan, David Taniar, Antonio Laganá, Youngsong Mun, and Hyunseung Choo, editors, *Computational Science and Its Applications - ICCSA 2006*, pages 422–431, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-34080-5.

[10] Emmanuel Jeannot and Guillaume Mercier. Near-optimal placement of mpi processes on hierarchical numa architectures. In Pasqua D'Ambra, Mario Guarracino, and Domenico Talia, editors, *Euro-Par 2010 - Parallel Processing*, pages 199–210, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg. ISBN 978-3-642-15291-7.

[11] Richard Earl Korf. Multi-way number partitioning. In *Twenty-First International Joint Conference on Artificial Intelligence*, 2009.

[12] Lalin L. Laudis, Shilpa Shyam, V. Suresh, and Ajay Kumar. A study: Various np-hard problems in vlsi and the need for biologically inspired heuristics. In Pankaj Kumar Sa, Sambit Bakshi, Ioannis K. Hatzilygeroudis, and Manmath Narayan Sahoo, editors, *Recent Findings in Intelligent Computing Techniques*, pages 193–204, Singapore, 2018. Springer Singapore. ISBN 978-981-10-8636-6.

[13] Eugene L. Lawler. The quadratic assignment problem. *Management Science*, 9(4): 586–599, 1963. doi: 10.1287/mnsc.9.4.586. URL `https://doi.org/10.1287/mnsc.9.4.586`.

[14] Andrea Lodi, Silvano Martello, and Michele Monaci. Two-dimensional packing problems: A survey. *European Journal of Operational Research*, 141(2): 241 – 252, 2002. ISSN 0377-2217. doi: https://doi.org/10.1016/S0377-2217(02)00123-6. URL `http://www.sciencedirect.com/science/article/pii/S0377221702001236`.

[15] A. R. Mamidala, Lei Chai, Hyun-Wook Jin, and D. K. Panda. Efficient SMP-aware MPI-level broadcast over InfiniBand's hardware multicast. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, page 8 pp., 2006. doi: 10.1109/IPDPS.2006.1639562.

[16] Silvano Martello, Michele Monaci, and Daniele Vigo. An exact approach to the strip-packing problem. *INFORMS Journal on Computing*, 15(3):310–319, 2003. doi: 10.1287/ijoc.15.3.310.16082. URL `https://pubsonline.informs.org/doi/abs/10.1287/ijoc.15.3.310.16082`.

[17] Guillaume Mercier and Emmanuel Jeannot. Improving mpi applications performance on multicore clusters with rank reordering. In Yiannis Cotronis, Anthony Danalis, Dimitrios S. Nikolopoulos, and Jack Dongarra, editors, *Recent Advances in the Message Passing Interface*, pages 39–49, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg. ISBN 978-3-642-24449-0.

[18] Grégory Mounié, Christophe Rapine, and Denis Trystram. Efficient Approximation Algorithms for Scheduling Malleable Tasks. pages 23–32. Association for Computing Machinery, 1999. URL `https://hal.archives-ouvertes.fr/hal-00001525`.

[19] MPI-Forum. *MPI: A Message-Passing Interface Standard. Version 3.1*, June 4th 2015. `www.{MPI}-forum.org`.

[20] Christoph Niethammer and Rolf Rabenseifner. An MPI interface for application and hardware aware cartesian topology optimization. In *Proceedings of the 26th European MPI Users' Group Meeting*, EuroMPI '19, pages 6:1–6:8, New York, NY, USA, 2019. ACM. ISBN 978-1-4503-7175-9. doi: 10.1145/3343211.3343217. URL `http://doi.acm.org/10.1145/3343211.3343217`.

[21] François Pellegrini and Jean Roman. Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In Heather Liddell, Adrian Colbrook, Bob Hertzberger, and Peter Sloot, editors, *High-Performance Computing and Networking*, pages 493–498, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-49955-8.

[22] Emil L. Post. Recursively enumerable sets of positive integers and their decision problems. *Bull. Amer. Math. Soc.*, 50(5):284–316, 05 1944. URL `https://projecteuclid.org:443/euclid.bams/1183505800`.

[23] Gerald Roth, John Mellor-crummey, Ken Kennedy, and R. Gregg Brickner. Compiling stencils in high performance fortran. In *In Supercomputing '97: Proceedings of the 1997 ACM/IEEE conference on Supercomputing*, pages 1–20. ACM Press, 1997.

[24] Y. Saad. *Iterative Methods for Sparse Linear Systems*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2nd edition, 2003. ISBN 0898715342.

[25] Sartaj Sahni and Teofilo Gonzalez. P-complete approximation problems. *J. ACM*, 23(3):555–565, July 1976. ISSN 0004-5411. doi: 10.1145/321958.321975. URL `http://doi.acm.org/10.1145/321958.321975`.

[26] Varun Sanduja and Rajeev Patial. Sobel edge detection using parallel architecture based on fpga. *International Journal of Applied Information Systems*, 3(4):20–24, 2012.

[27] Kirk Schloegel, George Karypis, and Vipin Kumar. Parallel static and dynamic multi-constraint graph partitioning. *Concurrency and Computation: Practice and Experience*, 14:219–240, 03 2002. doi: 10.1002/cpe.605.

[28] Uwe Schöning. Graph isomorphism is in the low hierarchy. *Journal of Computer and System Sciences*, 37(3):312 – 323, 1988. ISSN 0022-0000. doi: https://doi.org/10.1016/0022-0000(88)90010-4. URL `http://www.sciencedirect.com/science/article/pii/0022000088900104`.

[29] W. Tan, P. Lin, B. Liang, and H. Deng. Influence of network bandwidth on parallel computing performance with intra-node and inter-node communication. In *2009 Second International Conference on Intelligent Networks and Intelligent Systems*, pages 534–537, 2009. doi: 10.1109/ICINIS.2009.142.

[30] Hassler Whitney. Congruent graphs and the connectivity of graphs. *American Journal of Mathematics*, 54(1):150–168, 1932. ISSN 00029327, 10806377. URL `http://www.jstor.org/stable/2371086`.