MASTERARBEIT / MASTER'S THESIS

Titel der Masterarbeit / Title of the Master's Thesis

„Process Reordering in `MPI` for Cartesian Topologies and Isomorphic Communication Neighborhoods"

verfasst von / submitted by

Konrad von Kirchbach

angestrebter akademischer Grad / in partial fulfilment of the requirements for the degree of

Master of Science (MSc)

Wien, 2019 / Vienna, 2019

Studienkennzahl lt. Studienblatt / degree programme code as it appears on the student record sheet:	A 066 910
Studienrichtung lt. Studienblatt / degree programme as it appears on the student record sheet:	Masterstudium Computational Science
Betreut von / Supervisor:	Prof. Dr. Jesper Larsson Träff, MSc
Mitbetreut von / Co-Supervisor:	Dipl.-Math. Dipl.-Inform. Dr. Christian Schulz

Abstract

In this thesis, we introduce three new approaches for a one-to-one mapping of Cartesian graphs C with isomorphic communication (also called stencils) onto a symmetric hardware hierarchy. First, we present two greedy approaches, one centralized, one distributed, both using a priority queue to assign ranks onto the computation nodes, trying to maximize the number of on-node neighbors. The third approach, the hyperplane algorithm, recursively partitions the Cartesian graph C into two parts, while trying to minimize the communication between each partition, until all partitions can be mapped onto the computation nodes. The big advantage of the hyperplane algorithm is the fast runtime, which is no longer directly dependent on the number of vertices n or edges m in C.

We compared our approaches to `MPI`'s default rank assignment, `MPI`'s reordering scheme in the function `MPI_Cart_create`, Gropp's approach `Nodecart` and the general mapping tool VieM for different communication patterns or stencils. The hyperplane algorithm outperformed on average both of `MPI`'s approaches and outperformed or matched `Nodecart` for all stencils on the tested instances. For the general five-point stencil it was even capable of finding on average partitions with less inter-node communication than VieM. Both of the greedy approaches are only beneficial for certain communication patterns, but performed poorly for the general five-point stencil. We benchmarked the time needed for a `MPI_Neighbor_alltoall` exchange and could show that the reordering schemes presented in this thesis improved the bandwidth for various message sizes. For the sake of completeness, we measured and compared the instantiation time of all methods.

Zusammenfassung

In dieser Arbeit, stellen wir drei neue Herangehensweisen für eine eins-zu-eins Zuweisung von Cartesischen Graphen C mit einer isomorphischen Kommunikationsstruktur (auch bekannt als Stempel) auf eine symmetrische Hardwarehierarchie vor. Zuerst präsentieren wir zwei greedy Methoden, die eine zentralisiert, die andere verteilt. Beide Methoden benutzten eine Priority Queue um die Ränge den Knoten zuzuteilen und versuchen dabei die Kommunikation auf den Knoten zu maximieren. Als Drittes, stellen wir den Hyperplane Algorithmus vor, der den Inputgraphen rekursiv in zwei Teile teilt und dabei versucht die Kommunikation zwischen den Partitionen zu minimieren. Das wiederholt er, bis alle Partitionen auf die Knoten verteilt werden können. Der große Vorteil des Hyperplane Algorithmus ist, dass die Laufzeit nicht mehr von der Anzahl der Knoten n oder der Kanten m in C abhängt.

Wir vergleichen die vorgestellten Methoden mit `MPI`'s standard Rangverteilung, mit der Verteilung die durch das setzen der reordering Flagge in der Funktion `MPI_Cart_create` erlangt wird und der allgemeinen Mappingsoftware VieM für unterschiedliche Kommunikationsmuster oder Stempel. Der Hyperplane Algorithmus findet im Schnitt immer bessere Partitionen als die `MPI` Algorithmen und ist mindestens genauso gut wie der `Nodecart` Ansatz, für die getesteten Instanzen. Im Falle des generellen fünf-punkt Stempels, findet er sogar im Schnitt bessere Mappings als VieM. Die beiden greedy Algorithmen konnten nur für bestimmte Kommunikationsmuster vorteilhafte Partitionen finden. Wir haben die Kommunikationszeit für die `MPI_Neighbor_alltoall` Routine mit unterschiedlich großen Nachrichten gemessen und konnten zeigen, dass die Rangzuweisungsstrategien, die in dieser Arbeit präsentiert wurden einen positiven Effekt auf die Kommunikationsdauer hat. Schlussendlich haben wir auch die Instantiierungszeit für die unterschiedlichen Methoden gemessen.

Acknowledgements

First of all, I would like to specially thank my supervisors Prof. Dr. Jesper Larsson Träff and Dr. Christian Schulz for their support and giving me the opportunity to work on this project. Without their encouragement and advice, the process of writing this thesis would have been far less enjoyable. Thank you, Jesper, for the office, the possibility to ask you whenever I had questions and the great literature recommendations. Thank you, Christian, for introducing me to the whole topic of algorithms and being an excellent teacher. I would also like to thank Markus Lehr for proofreading my thesis.

I would also like to thank everybody who helped and supported me during the creation of this thesis. That is mainly my soon-to-be wife Simone Bachleitner, who gave me support and motivation whenever I needed it. Wolfgang Ost for the continuous talks on interesting topics during the lunch breaks and my family, without which none of this would have been possible. Thank you!

Contents

Chapter 1

Definitions

In this chapter we will present the definitions and terminology used throughout this thesis. The goal is to give the reader a necessary vocabulary for the following chapters. We will first explain the theoretical terminology and proceed to give a brief summary of important `MPI` functionalities and routines.

1.1 Theoretical Definitions

This thesis treats mapping for Cartesian graphs. In order to understand what a Cartesian graph is, we will first introduce the general concept of a graph, followed by what we define to be a grid and proceed to the definition of Cartesian graphs. In Chapter 2, we formulate the mapping problem as a quadratic assignment problem (QAP), hence we finish this section by introducing QAP in its original form.

1.1.1 Graphs

A *graph* $G = (V, E)$ consists of a set of *vertices* $V = \{0, \ldots, n-1\}$ and a set of *edges* $E \subseteq V \times V$. The vertices and the edges can be associated with weights given by some cost functions $c : V \to \mathbb{R}$ and $\omega : E \to \mathbb{R}$. We can write an edge as the pair of a source vertex $u \in V$ and a target vertex $v \in V$ i.e., (u, v). We say a graph is *undirected* if for every edge $e = (u, v) \in E$ there is a reverse edge (v, u), and the weights of both edges are equal. A *subgraph* $G' = (V', E')$ of G has a set of vertices $V' \subseteq V$ and a set of edges $E' \subseteq E \cap (V' \times V')$. If there is an edge $e \in E'$ between any two vertices $u, v \in V'$ in a subgraph G' of G then we call this subgraph a *clique*. The *neighborhood* of a vertex u is defined by all vertices that are adjacent from u, i.e., $N(u) := \{v \mid (u, v) \in E\}$.

1.1.2 Grids

A d-dimensional *grid* g is a tuple of *dimension sizes* $D = (d_1, \ldots, d_d)$ with $d_i \in \mathbb{N}^{+1}$ for $1 \leq i \leq d$ and a *basis* consisting of a set of orthonormal vectors $\mathscr{E} = \{\mathbf{e_1}, \ldots, \mathbf{e_d}\}$ defining the directions of the grid. For the sake of convenience, we can assume that each vector $e_i \in \mathscr{E}$ has only one non-zero entry (this can always be enforced by a suitable transformation of the reference coordinate system). A *point* x on a grid is defined by a *coordinate vector* $\mathbf{x} = (x_1, \ldots, x_d)$ in which it holds that $0 \leq x_i \leq d_i - 1$ for each dimension i. The set of all points on a grid is therefore given by $d_1 \times \cdots \times d_d$. We define the *size of a grid* $|g|$ to be the total number of points i.e., $|g| = \prod_{i=1}^{d} d_i$. We say a grid is *periodic* in a dimension i with $1 \leq i \leq d$ if first, the initial coordinate value x_i along that dimension is bound by 0 and $d_i - 1$, i.e., $x_i \in [0,\ d_i - 1]$ and secondly, the addition of any number $n \in \mathbb{Z}$ is defined to be $x_i + v := (x_i + v) \mod d_i$. A *subgrid* g' of a grid g has the same dimension d as the original grid and for each dimension p'_i with $1 \leq i \leq d$ of the subgrid holds that $1 \leq d'_i \leq d_i$. The set of points belonging to g' is a subset of the points on g.

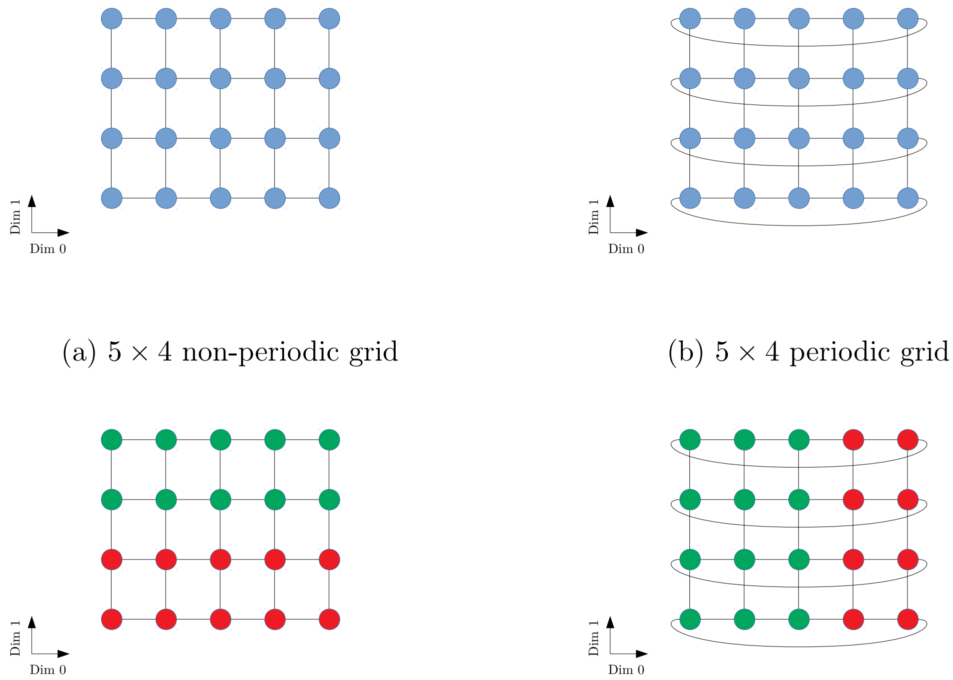
1.1.3 Cartesian Graphs

A *Cartesian graph* $C = (V, E)$ is a graph embedded on a d-dimensional grid. Each vertex $v \in V$ is associated with a point on the grid and has coordinates $(v_1, \ldots, v_d)$. For each vertex holds that $0 \leq v_i \leq d_i - 1$, for $0 \leq i \leq d_i$. For the sake of convenience, we assume that the number of vertices in a Cartesian graph C will always be equal to the number of points in a grid g, i.e., $|V| = \prod_{i=1}^{d} d_i$.

1.1.4 Quadratic Assignment Problem

The *quadratic assignment problem* was introduced in 1957 by Koopmans and Beckman [22]. The problem was to assign a set $\mathbb{F}$ of n facilities to a set $\mathbb{L}$ of n locations with the goal to minimize the total assignment cost, which consists of the total, weighted sum of flows. More formally, let $\mathbf{C} \in \mathbb{R}^{n \times n}$ be the flow intensity matrix, i.e., $C_{i,j}$ is the intensity of the flow between facilities i and j with $i, j \in \mathbb{F}$. Let $\mathbf{D} \in \mathbb{R}^{n \times n}$ be the *distance matrix*, i.e., $D_{i,j}$ is the distance between the locations k and l with $k, l \in \mathbb{L}$. Let $\Pi = \pi : \mathbb{N} \rightarrow \mathbb{N}$ be the set of all permutations π. A permutation π will assign facility $\pi(i) \in \mathbb{F}$ to entity $i \in \mathbb{L}$. Then we want to find a permutation $\pi^* \in \Pi$ that minimizes

[1] We define $\mathbb{N}^+$ to be the set of natural numbers without 0.



(a) 5×4 non-periodic grid

(b) 5×4 periodic grid

(c) 5×4 non-periodic grid, with subgrids

(d) 5×4 periodic grid, with subgrids

Figure 1.1: Two 5×4 grids. (a) is not periodic in any dimension. (b) is periodic along dimension 0. (c) two possible subgrids, green and red on the original non-periodic grid. (d) another two possible subgrids, green and red on the original periodic gird.

$$J(\mathbf{C}, \mathbf{D}, \pi) \coloneqq \sum_{i=1}^{n} \sum_{j=1}^{n} C_{\pi(i),\pi(j)} D_{i,j}. \tag{1.1}$$

We call $C_{\pi(i),\pi(j)} D_{i,j}$ the *weighted flow* between the facilities i and j. If the mapping is not strictly one-to-one, but multiple entities in $\mathbb{F}$ can be mapped to an entity in $\mathbb{L}$ then we define the load l of an entity $i \in \mathbb{L}$ to be the number of entities in $\mathbb{F}$ that are mapped to i, i.e., $l(i) = |\{j \in \mathbb{F} \mid \pi(j) = i\}|$.[2] In a one-to-one mapping all entities in $\mathbb{F}$ have a load of one. The problem can be generalized to many applications see [8, 43] for further reading.

[2]The function π is longer a permutation in this case.

1.2 Technical Definitions

All the algorithms presented in this thesis work within `MPI`, therefore it is imperative that the reader is aware of the `MPI` functions and routines used within these algorithms. For this sake, we will shortly introduce `MPI` and proceed to present the concepts of *communicators*, *virtual topologies* and communication routines and patterns, which are important for this thesis. This introduction is far from being complete, but we hope it will give the reader the necessary understanding for the later chapters.

1.2.1 Introduction to `MPI`

`MPI` [27] stands for *Message Passing Interface* and is a standardized API for distributed and parallel computing. In an `MPI` program a set of independent *processes* perform tasks in an MIMD or SIMD manner. The number of processes p for a run is specified by the user.

1.2.2 Communicators

In order to give context to the communication between processes, there exists an object called a *communicator*. A communicator represents an ordered group of processes. Each process can be identified within the communicator with a unique number called the *rank* R, for which it holds that $0 \leq R \leq p_{\texttt{comm}} - 1$, where $p_{\texttt{comm}}$ is the size of the group associated with the communicator, i.e., the number of processes in the communicator. `MPI` provides a default communicator for every run called `MPI_COMM_WORLD`, in which all processes are included. Processes within the group of a communicator can exchange point-to-point messages. One can extract the number of processes in a communicator with the function `MPI_Comm_size` and each process can retrieve its assigned rank R in the communicator with the routine `MPI_Comm_rank`, see Listing 1.1 for the function signature in C. The functions can also be used in C++.

Listing 1.1: Signatures of `MPI_Comm_rank` and the `MPI_Comm_size` routines

```
int MPI_Comm_rank(MPI_Comm comm, int *rank);
int MPI_Comm_size(MPI_Comm comm, int *size);
```

`MPI` provides a routine called `MPI_Comm_split` to create new communicators from an existing one. The group of processes associated with a new communicator is a subgroup of the group of processes in the old communicator. The groups of processes of the new communicators are mutually disjoint, i.e., no process in the original communicator is in two groups of the newly created communicators. Each process in the old communicator calling MPI_Comm_split can specify a number called the *color*, determining to which new communicator it will belong, and a number called the *key* determining the rank in the new communicator of the calling process. If a process passes the constant MPI_UNDEFINED as the color then the routine will return a construct called MPI_COMM_NULL, which is a handle for an invalid communicator, see Listing 1.2 for the function signature. The number of new communicators being created is defined by the number of distinct values of color passed by the processes in the old communicator. There exists a useful routine MPI_Comm_split_type which allows splitting by a type called MPI_COMM_TYPE_SHARED, which specifies that all processes in a new communicator have shared memory. This routine allows creating as many communicators as there are computation nodes in a run of the program.

Listing 1.2: Function signatures of MPI_Comm_split and MPI_Comm_split_type

```
int MPI_Comm_split(MPI_Comm comm,
        int color,
        int key,
        MPI_Comm *new_comm);

int MPI_Comm_split_type(MPI_Comm comm,
        int split_type,
        int key,
        MPI_Info info,
        MPI_Comm *new_comm);
```

MPI provides a function to get a mapping of the ranks in a communicator to the ranks of the processes in another communicator called MPI_Group_translate_ranks. Given the rank of the process in one of the communicators, the routine allows to compare the relative numbering of a process in the other communicator. The function takes an input the groups associated with the communicators, a number indicating the size of the smaller group and an integer array corresponding to the ranks of the smaller group. The output is an integer array with the corresponding ranks in the second group. See Listing 1.3 for the function signature.

Listing 1.3: MPI_Group_translate_ranks which finds a mapping between the ranks in group of one communicator to the ranks in group of another

```
int MPI_Group_translate_ranks(MPI_Group group1,
        int n, const int ranks1[],
        MPI_Group group2,
        int ranks2[]);
```

1.2.3 Virtual Topologies

Some communicators allow the processes to be arranged on a *virtual topology*, which is a special mapping of the processes associated with the communicator to some communication structure. A topology can be expressed as a graph $G = (V, E)$, where the processes correspond to the vertices V and the communication pattern between the processes can be expressed in the form of directed edges E. Note that even though there might not be a specified explicit communication edge between two processes, they can still communicate with one another.

A *Cartesian topology* is a regular d-dimensional grid that can be periodic along any dimension. A *Cartesian communicator* is a communicator in which the processes are arranged on a Cartesian topology. The tuple of dimension sizes $D = (d_1, \ldots, d_n)$ of the grid is defined by the number of processes along each dimension $d_i \in D$. A routine called `MPI_Dims_create` takes as input parameters the total number of processes p, the number of dimensions d and calculates a possible size of the grid, trying to make the size of each dimension as close to each other as possible. `MPI` has a routine to create Cartesian communicators called `MPI_Cart_create`, which take as input the dimension sizes D, a logical array specifying if a dimension is periodic and a flag allowing `MPI` to rearrange the processes to possibly better fit the physical machine. The number of processes p in a Cartesian communicator is given by $p = \prod_{i=1}^{n} d_i$. Each process with rank R in the Cartesian communicator is associated with coordinates $(R_1, \ldots, R_n)$ with $0 \leq R_i < d_i$, defining its position on the grid. There are `MPI` routines that allow to extract the coordinates of a process in a Cartesian communicator (`MPI_Cart_coords`), or get the rank associated with some specific coordinates (`MPI_Cart_rank`).

Listing 1.4: Signatures for `MPI_Cart_create`, `MPI_Dims_create`, `MPI_Cart_coord` and `MPI_Cart_rank`

```
int MPI_Cart_create(MPI_Comm comm_old,
        int ndims,
        const int dims[],
        const int periods[],
        int reorder,
        MPI_Comm *comm_cart);

int MPI_Dims_create(int nnodes, int ndims, int dims[]);

int MPI_Cart_coord(MPI_Comm comm, int rank, int maxdims,
        int coords[]);
int MPI_Cart_rank(MPI_Comm comm, int coords[], int *rank);
```

A *distributed graph topology* describes the communication relationship between processes in a general graph structure. The function `MPI_Dist_graph_create` allows to create a distributed graph communicator, taking as an input the adjacency list of communication neighbors, the weights of communication edges and a flag allowing `MPI` to reorder the processes, see Listing 1.5 for the function signature. Each process in the original communicator can specify its own communication neighbors and the weights of the communications, allowing for a scalable distributed way of specifying the global communication graph.

Listing 1.5: Function signature for the `MPI_Dist_graph_create` routine

```
int MPI_Dist_graph_create(MPI_Comm comm_old, int n,
        const int sources[], const int degrees[],
        const int destinations[], const int weights[],
        MPI_Info info, int reorder,
        MPI_Comm *comm_dist_graph);
```

1.2.4 Communication

This subsection introduces some terminology and routines for communication in `MPI`. We will present those we deem necessary to understand further parts of the thesis, in particular routines used for the experiments in Chapter 6.

Collective Communication

A communication pattern is called *collective* if it involves all processes in the group of a communicator. `MPI` provides several routines for collective communication such as, `MPI_Bcast` where a process, named root sends some data to all other processes in the communicator. `MPI_Scatter` provides the possibility for a rank, defined as root to send parts of some data to all other processes, whereas `MPI_Gather` each process in the communicator can send data to a designated process.

Listing 1.6: Function signatures of `MPI_Bcast`, `MPI_Scatter` and `MPI_Gather`

```
int MPI_Bcast(void * buffer, int count,
        MPI_Datatype datatype,
        int root, MPI_Comm comm);

int MPI_Scatter(const int *sendbuf, int sendcount,
        MPI_Datatype sendtype,
        void *recvbuf, int recvcount,
        MPI_Datatype recvtype,
        int root, MPI_Comm comm);

int MPI_Gather(const int *sendbuf, int sendcount,
        MPI_Datatype sendtype,
        void *recvbuf, int recvcount,
        MPI_Datatype recvtype,
        int root, MPI_Comm comm);
```

MPI also allows to aggregate the messages while sending it with a routine called MPI_Reduce. The aggregation can be done by different functions specified in MPI_Op. To name just a few, one can sum the values in the message up, extract the maximum, the minimum, multiply them or use logical con- and disjunctions. The aggregated message is collected by a user defined root process. Listing 1.7 provides the signature for the routine.

Listing 1.7: MPI_Reduce allows applying aggregation functions on the data while messaging it.

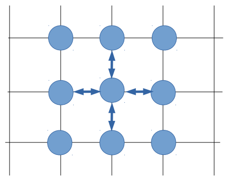
```
int MPI_Reduce(const void * sendbuf, void * recvbuf,
        int count, MPI_Datatype type, MPI_Op op,
        int root, MPI_Comm comm);
```

If a communicator has a virtual topology, one can use a routine called MPI_Neighbor_alltoall. Each process sends and receives the same amount of data from all its communication partners. The information about the communication partners for each process is stored in the communicator. For a Cartesian topology, the communication neighbors will be the two closest processes along each dimension, whereas for a distributed graph topology, the neighbors are simply the adjacent processes. The function signature can be found in Listing 1.8.

Listing 1.8: Function signature of the MPI_Neighbor_alltoall routine

```
int Neighbor_alltoall(const void * sendbuf,
        int sendcount, MPI_Datatype,
        void *recvbuff, int recvcount,
        MPI_Datatype recvtype, MPI_Comm comm);
```

Those are just a few examples of the collective operations MPI provides.


(a) five-point stencil in 2 dimensions.

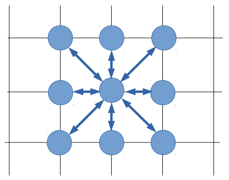

(b) nine-point stencil in 2 dimensions.

Figure 1.2: Examples of the five-point (a) and nine-point (b) stencil

Synchronization

For some applications, it is important to synchronize the processes in a program at certain points, for example when measuring the time needed for a subroutine. `MPI` provides the user with such a function called `MPI_Barrier`, which takes as only input a communicator. The barrier function terminates after all processes have entered it.

Listing 1.9: `MPI_Barrier` function signature.

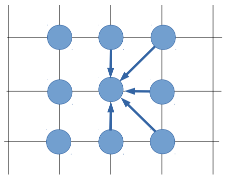
```
int MPI_Barrier(MPI_Comm comm);
```

Stencils

Träff et al. [42] introduced the term *isomorphic communication*. It is a communication pattern in Cartesian topologies specified for each process, by the same set of relative coordinate vectors. The isomorphic communication pattern can be expressed by a *stencil*, which is a set of vectors $\mathscr{N} = \{\mathbf{r_1}, \ldots, \mathbf{r_k}\}$. Each vector $\mathbf{r_i} \in \mathscr{N}$ specifies the offset in each dimension $(r_{i,0}, \ldots, r_{i,n})$ from the process.

A *five-point stencil* in 2 dimensions is the set of vectors describing the 4 closest neighbors, in terms of the Manhatten distance and the point itself. A *nine-point stencil* in 2 dimensions is the set of vectors describing the 8 closest neighbors and the point itself. The five- and nine-point stencils can be seen Figure 1.2. We define a *general five-point* or *nine-point stencil* to be a stencil in d dimension with the closest $2d$ or $3^d - 1$ neighbors and the point itself.

A *component* stencil is a stencil which generates different components in the Cartesian graph. On a 2D grid, each process would communicate with its closest neighbors along the first dimension, see Figure 1.4a. In 3D, each process communicates with its two closest neighbors in the first and second dimension, but no communication is performed in the last dimension. In general, each process communicates with it's 2 closest neighbors in each dimension from d_1 up to d_{n-1}.


(a) Crank-Nicolson stencil in 2 dimensions.

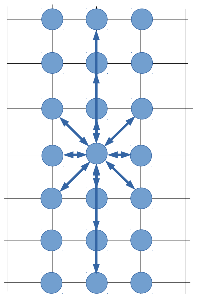

(b) Five-point stencil in 2 dimensions with two hops along the first dimension.

Figure 1.3: Example of the Crank-Nicolson (a) and five-point stencil with 2 hops in the first dimension (b) in 2D.

For the *diagonal* stencil, the communication neighbors are only along the diagonals. To be more specific, the relative neighbors are given by the set of all permutations of $\{-1, 1\}$ in the dimensions of the relative coordinate vectors, see Figure 1.4b for a 2D example.

A stencil, which is an often used scheme in finite difference for solving partial differential equations, is the *Crank-Nicolson* stencil [9]. It is essentially the same communication pattern as in the component stencil, but with one addition. Each process also communicates in the increasing direction of last dimension with same pattern as in the first $n-1$ dimensions, i.e., the set of relative coordinate vectors is duplicated and in the last coordinates of the duplicates, we set the original value of 0 to 1, see 1.3a for a 2D example.

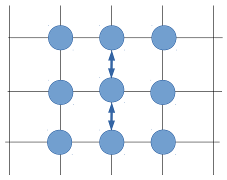
We define a *hop* to be a set $\mathscr{H} \subseteq \mathscr{N}$ of relative coordinate vectors $\mathbf{r_h}$, that have the same orientation but different sizes. To be more precise, let $\mathbf{r_i}, \mathbf{r_j} \in \mathscr{H}$, then it holds that

$$\mathbf{r_i} \cdot \mathbf{r_j} = ||\mathbf{r_i}|| \cdot ||\mathbf{r_j}|| \quad \text{and} \quad ||\mathbf{r_i}|| \neq ||\mathbf{r_j}||. \tag{1.2}$$

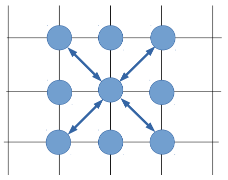
That is, the process has multiple communication partners in the direction given by some specific direction. See Figure 1.3b for a five-point stencil in 2D with 3 hops in the first and the last dimension.

Inter-node and Intra-node Communication

A *computation node* is a set of computation units that share memory. It can be composed of multiple sockets, each socket begin able to hold multiple cores. If



(a) Component stencil in 2 dimensions.



(b) Diagonal stencil in 2 dimensions.

Figure 1.4: Examples of a component (a) and diagonal (b) stencil in 2D.

two cores sit on the same node, they exchange data by using the shared memory. This process is called *intra-node* communication. If the two data-exchanging processes sit on different computation nodes of the machine, we call that kind of exchange *inter-node* communication.

Chapter 2

Introduction

Now that we have given the reader the necessary vocabulary, we can continue in this chapter, by presenting the motivation for the topic of this thesis. Further, we will propose a possible, formulation of the presented problem and conclude by giving an overview of the structure and the remaining chapters of the thesis.

2.1 Motivation

In parallel computation applications, a main bottleneck is the communication between computation nodes in the network. The communication on a computation node is multiple times faster than inter-node communication, see Figure 2.2 or Figure 2.1. That is why one should consider a good assignment of processes to the computation nodes in order to achieve high performance.

We want to map processes with intensive communication as closely as possible in the hardware hierarchy, ideally to the same computation node while maintaining the load balance between the nodes. Unfortunately the mapping problem is NP-Complete and has been shown to be equivalent to the graph embedding problem by Bokhari [4]. Thus, unless P=NP, we can only rely on good heuristics for large instances. Much effort was put into good mapping software such as VieM [34] and Scotch [31]. Most of these mapping software have as input the communication relationship between the processes as a general graph. The developers of Scotch claim that the runtime is linear in the number of edges in the communication graph and logarithmic in the amount of nodes in the distance graph. If the communication graph can be modeled as a much more structured Cartesian graph C and the distance graph D is symmetric and hierarchical while the communication is isomorphic then we can perhaps hope to find a scalable mapping heuristic. The highly structured problem may allow for an algorithm

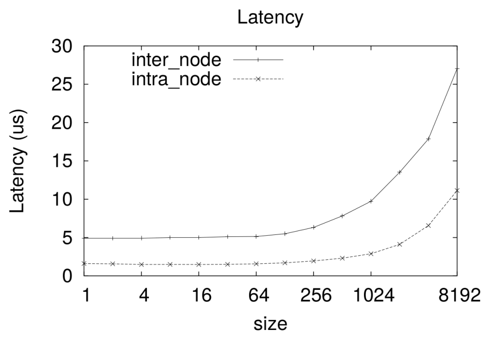


Figure 2.1: Inter- vs. intra-node communication performance. In the figure is shown the discrepancy between the inter- and intra-node latency for point-to-point communication. Figure taken directly from Mamidala et al. [24]

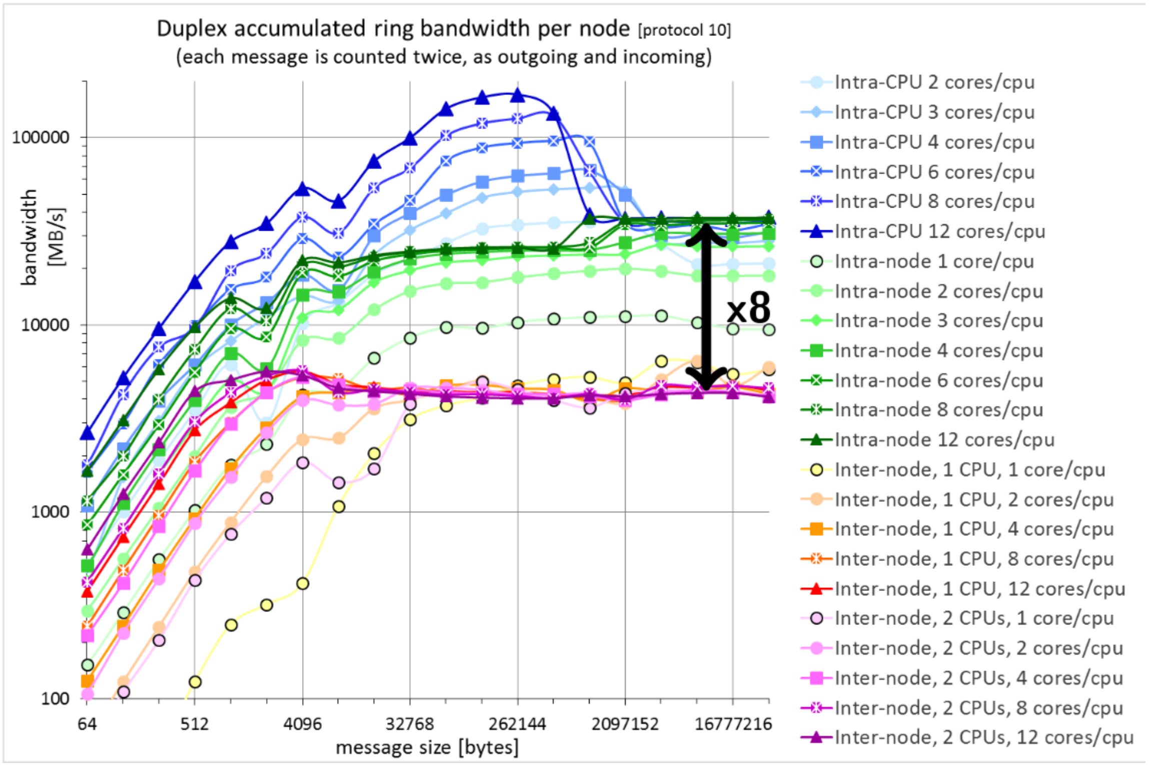


Figure 2.2: Different accumulated duplex ring bandwidth for communication on a computation node and different communication patterns. Depicted are intra-CPU (core-to-core and CPU-to-CPU) inter-node ring communication bandwidths. Figure taken directly from Niethammer and Rabenseifner [29].

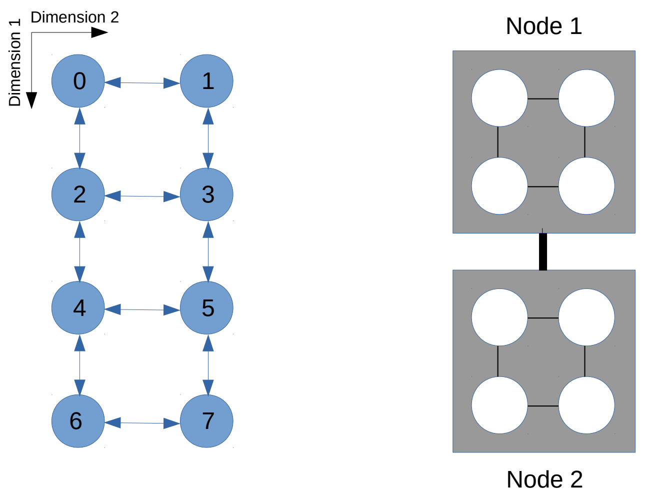


Figure 2.3: Example of a 4×2 Cartesian graph with a 5-point stencil and two computation nodes with 4 cores.

that is not dependent on the number of vertices or edges in C, but instead on a parameter that is typically smaller such as the dimension of the grid or the number of partitions, while still producing satisfying results. Cartesian communication graphs find a wide application in solving partial differential equations with finite difference methods, see [1, 37]. `MPI` allows for processes to be reordered when instantiating Cartesian and distributed graph communicators. As we will see, the reordering functions of `MPI` do not yield any improvement over the default consecutive order of the processes onto the nodes. In general, one can formulate the mapping problem as a quadratic assignment problem (QAP), see Subsection 1.1.4 in Chapter 1 by modeling the communication of the processes as a graph C and the distances between the cores as a graph D. Minimizing the amount of inter-node communication is equivalent to minimizing the objective function described in Equation(1.1).

2.2 Statement of the Problem

In this thesis we will look at reordering strategies in `MPI` of unweighted, directed Cartesian graphs with isomorphic communication patterns. In a Cartesian graph C, processes are represented by the vertices, while the communication edges for each process are given by a stencil. A directed edge $(u,\ v)$, representing processes u and v, implies a source vertex u sending a message to the target vertex v. We will ignore the details of the hardware interconnect, and assume that the communication cost between each computation node is equal and more expansive than the inter-node communication. When modeling the quadratic assignment problem, we will assume that the distance between two cores i and j is zero, if they are on the same computation node $D_{i,\ j} = 0$, and one if they are on separate computation nodes $D_{i,\ j} = 1$. With this choice of parameters, the value of the objective function defined in Equation(1.1) in Chapter 1 is exactly the number of inter-node communication. We want to find a permutation π of the processes in C that determines the new rank of i that is, $\pi(i)$ in order to minimize $J(C,\ D,\ \pi)$. We will focus on experimental performance in terms of inter-node communication reduction of the algorithms and their runtime. To be more precise, we will not give theoretical upper-bounds for the deviation of the obtained solution from the optimum, i.e., we will not develop approximation algorithms nor give a guarantee about the theoretical quality. The main objective is to devise an algorithm that can improve on Gropp's [13, 14] reduction of inter-node communication, for arbitrary isomorphic communication patterns, while still have a decent runtime, ideally not dependent on the number of vertices and edges of the input graph C. The algorithms should scale well and be easily portable to work on any multi-node machine running `MPI`.

2.3 Structure of Thesis

In Chapter 1 we give some basic definitions for graphs, grids, the quadratic assignment problem and features of `MPI`. Related work is presented in Chapter 3, along the explanation of an important algorithm of Gropp [14] representing the baseline for our approaches. Greedy approaches to the problem are discussed in Chapter 4, while a recursive approach is presented in Chapter 5. The comparison and evaluation between the different algorithms presented in this thesis is shown in Chapter 6. We conclude the thesis in Chapter 7 and give some perspective for future work.

Chapter 3

Related Work

In this chapter we present relevant research done on process mapping and rank reordering techniques. First we give try to give the reader an overview of process mapping approaches [15, 25, 26, 29, 36] developed specifically for or within `MPI`. We proceed by giving a short summary of general graph mapping techniques [5, 34] and communication aware partitioning using another library called CHARM++ [3]. The Vienna Mapping tool [34] (VieM) is described in more detail, since we will compare the amount of inter-node communication obtained by the algorithms developed in this thesis and VieM in Chapter 6, in order to get an estimate on the performance of the developed approaches in comparison to a general graph mapping software. We conclude this chapter with a detailed explanation of the `Nodecart` approach developed by Gropp [13, 14] which forms the motivating baseline for the development of this thesis. We include the `Nodecart` routine in all the experiments done in Chapter 6 in order to see, if the communication aware mappings in this thesis can improve on Gropp's approach.

3.1 Mapping with `MPI`

In this section, we will introduce some process mapping approaches developed for or within `MPI`. The goal is to give the reader a small overview of some research that has been done in order for him or her to see the benefits of the work developed in this thesis. The approaches of Mercier and Jeannot [25] and Subramoni et al. [36], described in the Subsections 3.1.1 and 3.1.2 differ to the problem in this thesis in the regard that they combine the extraction of the communication patterns and the process mapping, i.e., the communication pattern is not known in advance. Mirsadeghi and Afsahi [26], described in Subsection 3.1.3 exploits the pattern of collective communication algorithm, while Niethammer and Rabenseifner[29], described in Subsection 3.1.4 give the user the possibility to decompose the communication patterns in the basis of a Cartesian grid.

3.1.1 Distributed Graph Communicator Mapping

The approaches described in this section were developed for arbitrary virtual topologies, i.e., unstructured graphs. We decided to include them, since our greedy approaches presented in Chapter 4 are not restricted to a Cartesian topology and hence, could also be applied to distributed graph topologies.

Hatazaki

Hatazaki [15] proposes an algorithm for the graph embedding problem of an unweighted communication graph G onto the hardware topology, represented as a complete host graph H (there exists a path between each pair of vertices). The main contribution is the leveraging of symmetry in the host graph H. Similarly to our approach, Hatazaki addresses the problem as finding a permutation π that finds a sufficiently small value for the objective function defined in Equation (1.1), describing the weighted communication. Note that with unit edge weight, the objective function is reduced to the sum of distances between cores onto which processes have been mapped.

Hatazaki exploits the fact that processes on the same hardware level have the same communication distance. With that assumption, the host graph H can be modelled as a host tree. The leaf vertices of the host tree correspond to a list of entities on the highest level of the hardware hierarchy, having the same pairwise communication distance. Internal vertices store the number of subentities in the leaves of their subtree. Edges of the tree are unweighted.

The mapping algorithm starts by assigning all processes to the root vertex of the host tree. The processes are partitioned into k blocks, where k denotes the number of children of the root vertex s.t. the sum of inter-block edges is as small as possible. Hatazaki adopted the Kernighan-Lin heuristic [12, 21] for improving the partition of the communication graph. This step is repeated recursively to the blocks and subtrees of the root's children, until all processes have been assigned to the leaves of the host tree.

Hatazaki evaluated his approach by measuring the performance in communication for a five-point stencil exchange on a Cartesian grid. The machine was able to host up to 1024 processes. He measured the reduction of inter-node communication compared to the default order of `MPI` and could obtain values of up to 75%. Further, Hatazaki was able to reduce the time needed for the communication up to 90% for message sizes of 100 000 B.

Träff

Träff [40] implemented an algorithm specifically for the NEC SX-series of parallel vector computers. In his model, the target machine is described by an undirected, complete, weighted host graph H and an undirected, weighted communication graph C. Assuming that the machine has a hierarchical communication pattern, the host graph H can be described as a tree. The leaves correspond to processors, while internal vertices denote the communication distance between the processors in different subtrees.

Träff depicts the equivalency between the *graph embedding problem*, for unweighted guest graphs on weighted host graphs and the *graph partitioning problem*. He does this, by showing that the embedding problem is special case of the weighted graph partitioning problem with a special correspondence of the objective functions. Thus, by finding an optimal solution to the graph partitioning problem, he solves the graph embedding problem to optimality. Therefore, he can solve the graph embedding problem for a communication graph C onto a hierarchical system by recursively partitioning C into the number of entities on each hierarchy level.

Träff uses Kernighan-Lin-like local-search technique [41] to improve on an initial k-partition, in which the process of finding a sequence of vertex moves is no longer dependent on the number of blocks k. The algorithm was designed not only to reduce the total-cut, but also to ensure that the max-cut (the bottleneck) can only be decreased. Even though Träff's approach is centralized, some aspects of the local-search techniques can be vectorized.

The evaluation of the mapping is done on Cartesian and a variety of general graphs. The communication time for a neighbor data exchange routine was measured and Träff could show that the reordering schemes and the communication time was smaller than for virtual topologies without reordering. For some general graph instances, he could even measure an increase of communication time to a factor of almost nine.

Mercier and Jeannot

Mercier and Jeannot [25] present a technique for rank reordering specifically for distributed graph topologies in `MPI`. Even though their approach is very flexible, it is fully centralized. The algorithm works as follows: First a pre-run of the `MPI`-application has to be done, in order to gather information about the communication patterns between the processes. From the communication information, a communication graph can be built. Next, they gather information about the hardware using a library called HWLOC [7] and store this information in a hierarchical tree representing the distances between cores. The root can then proceed to calculate a process reordering based on the *TreeMatch* algorithm [18]. The TreeMatch algorithm works in a recursive, bottom-up manner, i.e., from the cheapest to most expansive level in the communication hierarchy, by grouping the processes on each hierarchy level in a way that minimizes the remaining communication on the upper levels of the hardware hierarchy.

They measure the improvement of the proposed technique with two experiments on a cluster with 8 nodes, having 8 cores each. In the first experiment, 8 rings consisting of 8 processes exchange data in a circular manner, i.e., within a ring, a designated process sends data to a neighbor which in turns sends it to its neighbor and so on until the starting process receives the data again, after which the rings exchange the data over a designated process in each of them. They reported an improvement of 10 to 20% compared to the standard, consecutive `MPI` mapping in communication time, both in terms of data size and number of messages send. The second experiment is based on a real application of astrophysics involving computational fluid dynamics code called ZEUS-MP [16]. It involves a three-dimensional computation domain, which is decomposed into tiles where boundary elements exchange data. ZEUS-MP originally uses a Cartesian communicator without the reordering flag (with the standard rank ordering). Mercier and Jeannot measured communication performance improvements of up to 15%.

3.1.2 Switch Traversal Minimization

Subramoni et al. [36] developed a two-step mapping tool in order to minimize the number of switch traversals in a hardware interconnect and maximizing the amount of intra-node communication. They first retrieve the amount of switch traversals for each node pair using utilities from the OFED distribution [30]. The more switch traversals is needed between to end-points, the greater the communication latency. After gathering the hardware details, the *Neighbor-Joining* algorithm is used to construct a binary tree incorporating the distances between the computation nodes. Even though they focus on tree-based topologies, they mention that their idea is generalizable to other types of networks. The Neighbor-Joining method [32] was originally developed for the construction of phylogenetic trees in biology and is a bottom-up approach, joining nodes with the smallest distances until only one node is left. The binary tree is then constructed by attaching all node pairs that were joined to the node they created. Information about the communication patterns is gathered in a pre-run with profiling tools. A top-down approach is done to recursively perform the mapping with an external graph partitioner software, with the goal to minimize long distance communication over the network and maximizing inter-node communication cost. Since a single execution of a partitioning software may not yield a balanced, high quality mapping, multiple runs are done until such a mapping is achieved. In the paper, Subramoni et al. evaluate different aspects of the mapping tool. We will focus on the performance benefits for applications, as they are the most relevant to us. They first benchmarked the communication improvement of a general five-point stencil in 3D, with 512 processes and small (2000 B) to medium (32 000 B) data exchange sizes, showing improvements in message exchange time of up to 40% over the default mapping. The also tested the improvement of inter-node communication over the default `MPI` mapping for different applications, namely AWP-ODC [10], a model to simulate wave propagation in earthquakes, which also uses a five-point communication pattern in 3D and Hypre [11], which is an open-source software of high performance, parallel, linear equations solvers and preconditioners. Their approach improved the amount of intra-node communication from the standard rank order of 33% to 66% for AWP-ODC and from 45% to 53.3% for Hypre.

3.1.3 Reordering for Collective Communication

Mirsadeghi and Afsahi [26] propose topology aware rank reordering in `MPI` specifically for collective communication. More precisely, they exploit the communication pattern of collective, hierarchical algorithms such as *Recursive Doubling* and non-hierarchical algorithms such as *Ring* [38], as they are the most commonly used algorithms for `MPI_Allgather`, which is a collective routine that enables to share local data held by the processes across all processes.

Recursive Doubling works roughly as follows; processes communicate over $\log_2 P$ phases, where P is the number of processes in the communicator. In each phase t, process i exchanges data with process $i \oplus 2^t$ where $\oplus$ stands for the binary XOR operator. The amount of data that two processes exchange increases over the communication rounds. The ring algorithm runs in $P - 1$ rounds. In each round t, process i receives data from process $i - 1$ and sends data to process $i + 1$.

Since the communication partners are known, the reordering can be calculated without the specification of a communication graph. They use the HWLOC [7] library and InfiniBand tools to gather information about the distances between the computation nodes. Their approaches reorder the ranks in such a way that the pairs of processes exchanging the largest amount of data are positioned on nodes close to each other.

They benchmark the latency of the `MPI_Allgather` routines developed by them, from the standard `MVAPICH2` and from mappings obtained by the Scotch Graph Partitioning Library [31] with the OSU Mico-Benchmarks [39]. As initial rank order, they used four different schemes that are common and greatly impact the default performance of the `MPI_Allgather` algorithms. The four base communicators are built, using a consecutive or round-robin rank assignment on the nodes and on the sockets separately, i.e., consecutive rank assignment to the nodes and consecutive or round-robin assignment to the sockets or round-robin assignment to the nodes with consecutive or round-robin assignment to the sockets. They benchmarked with 4096 processes with a maximum of 256KB in message size. For non-hierarchical approaches, they improve the latency up to 67% for message sizes bellow 1KB and up to 78% for message sizes larger than 1KB, for different initial rank orders. They note that Scotch does not perform well compared to their and the default rank orders. For hierarchical approaches, they could show an improvement of up to 30% for message sizes below and above 1KB. A drawback of this method is that if the original ranks have potentially large data stored locally, a reordering of the ranks specifically for the broadcast operation can result in a lot of non-local data access or intensive communication between the original and the newly assigned ranks. For large message sizes, this operation outweighs all the gain obtained from the reordering approach.

3.1.4 Cartesian Communicator Mapping

Rabenseifner and Niethammer [29] address three problems in the current `MPI` Version for the mapping of processes in Cartesian communicators for applications with Cartesian *halo* (region to store data from other subdomains) communication patterns. First, they note that the dimension sizes created by `MPI_Dims_create` is independent of the actual application grid[1], i.e., it only tries to minimize the difference in dimension sizes, instead of minimizing the communication over the subdomains. Secondly, the factorization of the `MPI` processes is independent of the underlying hardware and finally they remark that the default consecutive mapping of `MPI` is suboptimal. Their approach solves the three problems combined, i.e., the domain decomposition is no longer independent of the hardware topology. For that purpose, they propose a multi-level approach. While they mentioned in their paper that their technique is generalizable to any number of hardware hierarchies, the highest layer presented in the paper is that of the computation nodes. Their algorithm can work his way down in the hardware hierarchy from the computation nodes, to the NUMA domains, up to the cores. Since the communication between the computation node is the slowest, they minimize it first, by finding a decomposition of the number of nodes into a Cartesian grid s.t. the surface between the domains is evenly balanced. To be more specific, they decompose the number of nodes N into a

$$N = \prod_{i=1}^{d} n_i \tag{3.1}$$

[1]The data grid.

Cartesian topology. They formalize a communication cost for exchange of halo regions for each subdomain with

$$c = 2 \sum_{i=1}^{d} c_i h_i \prod_{j=1 \& j \neq i}^{d} \frac{g_j}{n_j}, \tag{3.2}$$

where c_i is a communication cost factor in dimension i, h_i is depth of the halo in dimension i, while g_j and n_j describe the dimension sizes of the application grid and the node grid in dimension j. The intuition behind this formula is simple, it expresses the weighted surface of the subdomain that is communicating to other subdomains. Note that they assume the same communication cost factor in both directions of i, enforcing the approach to consider symmetric communication in the dimensions. With this formalization, they factorize the number of nodes N under the Constraints 3.1 and 3.2, while minimizing the latter.

The technique for the next hardware levels is the same, but instead of operating on the original grid, the mapping is done on the induced subgrids by the factorization of the step before, with the adapted communication cost function

$$c' = 2 \sum_{i=1}^{d} c_i h_i \prod_{j=1 \& j \neq i}^{d} \frac{g'_j}{p_j}, \tag{3.3}$$

where $g'_i = \frac{g_i}{n_i}$ and $P = \prod_{i=1}^{d} p_i$ is the number of cores per node. This multi-level approach requires each of the sublevels to be of equal size, i.e., the hardware interconnect should be symmetric.

Each process calculates its rank and coordinates on every hardware level passed by the algorithm to combine those ranks to find its new overall rank and coordinate in the original process grid. They hope to make this routine part of the `MPI`-Standard under the name `MPI_Cart_create_weighted`.

In order to find good domain decompositions, the user provides to the function `MPI_Cart_create_weighted` the communication weights in each dimension of the application grid or a constant indicating equal weights.

They compared the runtime of `MPI_Dims_create` combined with `MPI_Cart_create` to `MPI_Cart_create_weighted` with equal weights and data grid dimension size based weights for 3 dimensional grids with grid size ratios of $1 : 2 : 4$. They measured a decrease of the halo size of up to 32% and a reduction in communication time of up to 77.3%.

3.2 Other Mapping Approaches

In this section we will give a short overview of mapping techniques proposed by Brandfass, Alrutz and Gerhold which use the Müller-Merbach algorithm combined with local search on which Schulz and Träff improve on. For the sake of diversity, we include a mapping approach taken with another library called CHARM++. The first two approaches work for general graph structures, whereas the latter is designed for Cartesian topologies.

3.2.1 Brandfass, Alrutz and Gerhold

Brandfass, Alrutz and Gerhold [5] propose a slight modification to the Müller-Merbach algorithm [28] for assigning facilities to locations in the QAP problem, see Section 1.1.4 in Chapter 1 combined with the local search technique proposed by Heider [17]. The mapping of the communication graph onto the host graph is formulated as a QAP. Müller-Merbach takes as input a matrix containing the communication information between each pair of processes and a complete distance matrix containing the pairwise distances between the cores. In the first round, the total communication is computed for each process, as is the total distance for each core. The process with the maximal amount of communication is then mapped onto the core with the smallest total distance. In the next rounds, only the summed-up communication load between already assigned and non-assigned processes and the summed-up distances between assigned and unassigned cores is calculated, again assigning the process with the biggest communication load to the core with the smallest summed-up distance. Heider's approach belongs to the class of local search algorithms and takes in an initial feasible solution for the QAP. It tries to improve the solution with an exchange of assignments between pairs of processes i and $i+1$ in cyclic manner until no further improvement is found.

In order to reduce the runtime of the algorithm, Brandfass, Alrutz and Gerhold substitute the communication matrix C and the distance matrix D by symmetric matrices $C := \frac{1}{2}(C + C^T)$ and $D := \frac{1}{2}(D + D^T)$ since it does not change the outcome of the objective function, see Equation (1.1), but reduces the amount of arithmetic operations. This solution serves as the initial feasible input for the local search. Further, they neglect pairs of processes, for which the objective function does not change, i.e., processes that are assigned to the same computation node are not swapped. As a further improvement of the runtime, they reduce the search space of the algorithm by partitioning into subdomains and running the local search in each of the subdomains of the search-space successively. This of course limits the algorithm from finding solutions outside of the search-space.

They benchmark the approaches on the flow solver of the DLR TAU code [35] with varying input graph sizes. They first partition the input graph using different initial partitioning techniques (recursive coordinate bisection and graph partitioning), map the partitions using the Müller-Merbach scheme and use local search to improve the quality of the mapping. The compared the default, initial mapping of `MPI` to a mapping only using the Müller-Merbach scheme without local search and to Müller-Merbach with local search. While they note that the initial graph partitioning method has no big influence onto the improvement, their approach is able to find solutions which reduce the objective function value by 36% compared to the initial mapping.

3.2.2 Vienna Mapping

Schulz and Träff [33] developed mapping techniques based on Brandfass et al.. The authors assume sparse communication patterns and a symmetric hardware hierarchy, i.e., each entity in some level of the hierarchy as the same number of subentities. The hardware hierarchy can therefore be described using strings in the following way, let $s = e_1 : e_2 : \cdots : e_m$ be a string describing the m-level hierarchy of the underlying hardware system and $c = c_1 : \cdots : c_m$ be a string describing the distance between entities in different hardware levels. To be more precise, the string s describes how many entities are on each level of the hardware system, i.e., e_1 could denote the number of cores per NUMA domain, while e_2 denotes the number of NUMA domains on a node, e_3 the number of nodes and so on. The string c describes the distance between two entities that are in different subsystems. These assumptions allow to use a graph representation of the communication matrix C, since the communication matrix is now sparse. Further, it is not necessary to store the complete distance matrix, but instead one can use a tree representation to describe the distances between different cores.

An additional speed-up is obtained, by reducing the search space. This is done by only allowing swaps between processes that actually communicate with each other, i.e., only processes that are connected by an edge in the communication graph C. The authors extend the definition of the search space, by allowing swaps between processes having a distance of at most k in the graph theoretical sense, i.e., the length of the unweighted, shortest path between two processes is at most k. These assumptions and data-structures greatly reduce the time needed to calculate and update the objective-function value.

Instead of using Müller-Merbach to obtain an initial solution, the authors propose two different techniques, a *bottom-up* and *top-down* approach. The top-down approach works approximately by first partitioning the input communication graph C into e_m perfectly balanced parts and use mapping tools, combined with local search techniques in order to find a permutation π that minimizes the objective function value defined in Equation (1.1) in Chapter 1. In the next step, each of the e_m partitions are again partitioned and mapped into and onto e_{m-1} parts. This is done until all hierarchy levels have been mapped. The bottom-up approach works similarly, by first partitioning the input communication graph C into $|V|/e_1$[2] perfectly balanced parts and map those on a unique system entity being able to host e_1 entities. These blocks are contracted and the partitioning and mapping step is repeated until the last graph C' is partitioned and mapped onto $|V'|/e_m$ blocks.

In experimental evaluations, the presented techniques measured an average improvement of 52% in the objective function value using the top-down approach compared to Müller-Merbach, with the same local-search neighborhood. While this is very promising, they were also on average slower by a factor of 194 than Müller-Merbach, due to the expansive perfectly-balanced graph partitioning techniques.

3.2.3 Mapping with the CHARM++ Library

Bhatele and Kalé [3] use a library called CHARM++ [19, 20] to decompose the computation tasks into virtual processor objects called *chares*. The granularity of the virtual objects can be higher than of the computation tasks, i.e., there can be more chares than cores. The chares are stored in a *chare array*, which lies on top of the underlying data array of the problem. To be more specific, each chare is responsible for a connected part in the data array. The individual chares in the chare array are then mapped to the cores of the physical machine. A mapping of the chares can be specified by the user. The mapping technique suggested in the paper is specific for a five-point stencil in 2D or 3D. The main goal is to map as many communicating chares as possible to the same physical core while preserving load balance. For that purpose and under the assumption that communicating chares are located next to each other in the chare array, the array of chares is decomposed into equally sized boxes. Boxes that communicate intensively are mapped onto nearby processors. The authors compare their topology aware mapping for the five-point stencil in 2- and 3D against a round-robin mapping. In the best case, for a chare grid of size $16 \times 16 \times 16$, a data grid of size $1024 \times 512 \times 512$, 2048 processes and over 1000 data exchanges between communicating chare elements, the authors were able reduce the communication time by a factor of two over the round robin mapping.

[2]Recall that V is the set of vertices in a graph and therefore $|V|$ is the number of vertices

3.3 Gropp's Algorithm as the Baseline

Gropp developed an algorithm [13, 14] to create Cartesian communicators in `MPI` using node information. Gropp pointed out that passing the reordering flag to `MPI_Cart_create` does not yield any improvement over the default mapping of `MPI`. He proposed a simple solution; reordering the processes within the communicator to maximize inter-node communication, while ignoring the rest of the hardware interconnect. While not optimal, it yields significant improvements of the default mapping of `MPI` for many instances, with little need for communication between the processes. Two assumptions are made for the algorithm; first, each computation node has the same number of processes and second, the function `MPI_Comm_split_type` with type `MPI_COMM_TYPE_SHARED` creates communicators, where each of the processes in a group of the newly created communicator belong to the same node.

The algorithm works in the following way.

- Use the `MPI_Comm_split_type` function to identify the nodes and create a leader communicator, consisting of one process per node.
- Communicate to each process the number of nodes, i.e., the size of the leader communicator.
- Create a two-level decomposition of the grid, one level consisting of the node grid and the other being the grid of processes on the node. For that purpose, Gropp uses his own factorization which is explained below, since `MPI_Dims_create` will not always produce compatible grid decompositions.
- With the grid of the computation nodes and the grid of the processes on a node, each process can calculate its new rank without the need for communication.
- Given the new coordinates of the processes, a new communicator can be created with all processes specifying the color to be zero and their new rank as key.

The reason why Gropp uses his own factorization is that he needs to ensure that the dimensions of the node and the dimension of the processes per node grids are compatible with the dimensions given by the overall grid. To be more specific, for every dimension i the product of the node grid size in dimension i, $d_{\text{node, } i}$, and the process grid size in dimension i, $d_{\text{process, } i}$, must equal to the number of processes in dimension i, d_i of the original grid, i.e., $d_i = d_{\text{node, } i} \cdot d_{\text{process, } i}$. This constraint is not always fulfilled by the dimension sizes created by `MPI_Dims_create`. In order to achieve this, he factorizes the number of processes per node into primes and assigns those to the intra-dimensions (dimensions of the processes per node grid) while minimizing the sum of inter-node dimension sizes. The pseudocode for the two routines, taken directly from Gropp [14] can be found in Algorithm 2 and Algorithm 1.

As one can see from the pseudocode, no heavy computation is done so the algorithm runs very fast and scales well. In Figure 3.1, we can see an example of an optimal process reordering by Gropp, for a 4×4 grid and a five-point stencil. The nodes are represented by the grey rectangles.

While Gropp's algorithm provides good mappings for the general five- and nine-point stencil, it is oblivious to more general communication patterns. We want to improve on his idea, by incorporating a reordering criterion based for any isomorphic communication patterns given by a stencil. The hope is that we can exploit the structured communication patterns in order to avoid mappings that are worse than the default mapping. An example for a rank reordering by Gropp's algorithm that performs worse than the default mapping can be seen in Figure 3.2.

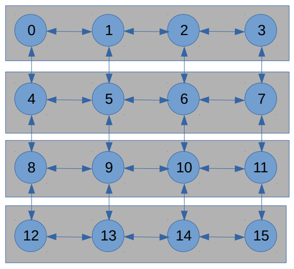
Gropp measured the performance of his approach with a general five-point stencil on a 2- and 3-dimensional mesh and compared it to the default mapping of `MPI`. The tests were performed on three different systems with up to 4096 processes. In each experiment, data of size up to 100KB were exchanged over 20 iterations. Each experiment was repeated thirty times. Gropp reported the average and the minimum running time for each experiment. He reported improvements in the computation rate (GFLOP/s) over the Cartesian communicator without reordering up to 11%.

Algorithm 1: 2-level decomposition

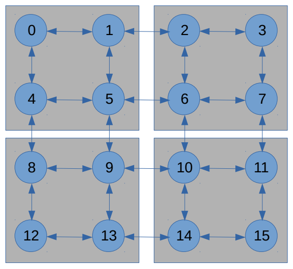
```
    Input: Dimension sizes $dims$,
    number of dimensions $ndims$,
    number of nodes $nnodes$
    Result: Dimensions of the processes on computation node grid
2-level decomposition($dims$, $ndims$, $nnodes$)
 1  $factors \leftarrow$ Prime factors of $nnodes$
 2  for $i \leftarrow 0$ to $ndims$ do
 3      intradims[i]$\leftarrow 1$
 4      remaindims[i]$\leftarrow dims[1]$
 5  end
 6  while $nnodes > 1$ do
 7      //Find largest prime factor and remove it from factors
 8      $fac \leftarrow$largest prime factors of $factors$
 9      //Find index $j$ s.t. $fac$ divides remaindims[j] and remaindims[j] is larger
          than any other remaindims[k], where $fac$ also divides remaindims[k].
10      intradims[j] $\leftarrow$ intradims[j]$\cdot fac$
11      remaindims[j] $\leftarrow \frac{\text{remaindims[j]}}{fac}$
12  end
```

Algorithm 2: `MPIX_Nodecart_create`

```
   Input: Old communicator $oldcomm$,
   dimension sizes $dims$,
   number of dimensions $ndims$
   Result: New communicator with Gropp's rearrangement $nodecart\ comm$
MPIX_Nodecart_create($oldcomm\ dims, ndims$)
 1 //Find the nodes and create node communicator $nodecomm$
 2 MPI_Comm_split_type($oldcomm$, MPI_COMM_TYPE_SHARED, $rank$,
     MPI_INFO_NULL, &$nodecomm$)
 3 //Create leader communicator
 4 MPI_Comm_rank($nodecomm$, &$nrank$)
 5 $color \leftarrow$ MPI_UNDEFINED
 6 if $nrank = 0$ then
 7 |   $color \leftarrow 0$
 8 end
 9 //Create the leader comm consisting of one process per node
10 MPI_Comm_split($oldcomm$, $color$, $oldrank$, & $leadercomm$)
11 //Broadcast the number of nodes
12 if $color = 0$ then
13 |   MPI_Comm_size($leadercomm$, &$nnodes$)
14 end
15 MPI_Bcast($nnodes$, 1, MPI_INT, 0, $nodecomm$)
16 //Calculate the grid sizes for the nodes and the processes
17 intradims $\leftarrow$2-level decomposition($dims, ndims, nnodes$)
18 interdims[i] $\leftarrow \frac{dims[i]}{\text{intradims[i]}}$
19 //Find coordinates in virtual grid
20 //Extract the intercoordinate from the rank in leadercomm
21 //Extract the intracoordinate from the rank in nodecomm
22 coords[i] $\leftarrow$ intracoords[i] + intercoords[i]$*$ intradims[i]
23 //Calculate the rank of calling process and create new communicator
24 $rr \leftarrow$ coords[0]
25 for $i \leftarrow 1$ to $numdims$ do
26 |   $rr \leftarrow rr \cdot dims[i]$+coords[i]
27 end
28 MPI_Comm_split($oldcomm$, 0, $rr$, & $nodecartcomm$)
```

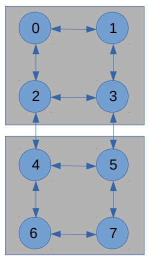


(a) MPI's default mapping.

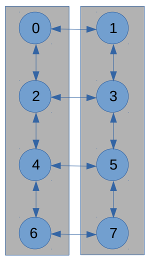


(b) Gropp's reordering strategy.

Figure 3.1: Assignment of 4×4 grid with a five-point communication pattern onto four nodes, with four cores each. The default mapping of MPI, (a) maps the processes consecutively onto the computation nodes, resulting in a total of 24 inter-node communication edges. Two nodes have the maximum of inter-node communication edges per node, which is 8. Gropp's rearrangement (b) results in a mapping, producing 16 inter-node communication edges in total and all nodes having 4 inter-node communication edges, thus improving over the default mapping.



(a) MPI's default mapping.



(b) Gropp's reordering strategy.

Figure 3.2: Assignment of 4×2 grid with a five-point communication pattern onto two nodes, with four cores each. The consecutive default mapping of MPI, (a) results in a total of 4 inter-node communication edges and two nodes having 2 inter-node communication edges. Gropp's rearrangement (b) on the other hand results in a mapping producing 8 inter-node communication edges in total and all nodes having 4 inter-node communication edges.

Chapter 4

Greedy Algorithms

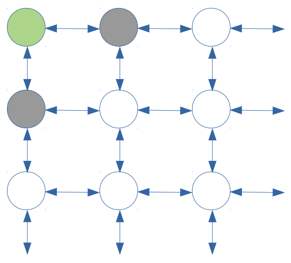
In this chapter, we will cover two greedy approaches that we developed to assign the processes to the computation nodes. We start with a very trivial, centralized greedy method in which we use a priority queue to assign ranks to the computation nodes. We proceed by extending that technique to a distributed approach, in which we find a set of processes, which is the size of the number of nodes N, in which each process is responsible to assign ranks to a node in parallel. Both of the methods presented assume that there is the same amount of processes p_{node} on each node. They could easily be adapted to handle various computation node sizes.

4.1 Centralized Greedy Approach

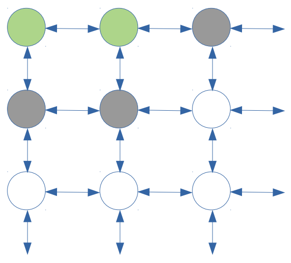
This approach does not take advantage of the Cartesian graph structure, nor the structure of the communication. It seems to be the straightforward thing to do for a general graph, and thus this approach as the next, can also be used for any topology. We will see in Chapter 6 that due to its obliviousness to the structure of the problem, the approaches here will perform worse than the method described in the next chapter.

4.1.1 Motivation

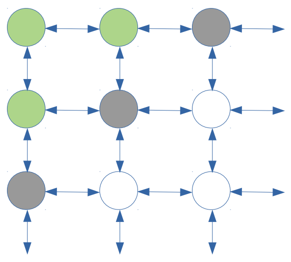
The first intuitive idea is to use a centralized approach to greedily assign the ranks to the computation nodes. The greedy step consists in irreversibly assigning a process with maximal number of neighbors already on the computation node to the node. This can easily be done with a priority queue. The priority of the ranks in the queue is determined by the number of its neighbors that are already assigned to the computation node and the number of neighbors it has in the queue. A tie is



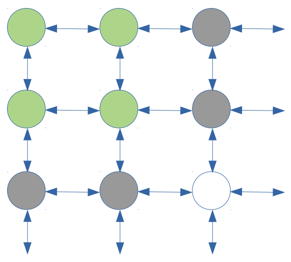
(a) Step 1



(b) Step 2



(c) Step 3



(d) Step 4

Figure 4.1: Motivating example for the priority of process assignments for a five-point stencil. (a): A first vertex (green) is assigned to the computation node. The neighbors of the assigned rank are enqueued (grey). (b): One of the vertices in the priority queue is assigned to the computation node and its neighbors are put into the queue. (c): We want to make sure that the vertex with the most neighbors on the computation node and in the priority queue is assigned next. (d): Assign again the vertex with the most neighbors on the computation node and in the queue to the computation node, trying to maximize the number of communication edges on the node.

broken by prioritizing the smaller rank, assuming that the ranks are initially mapped consecutively to the computation nodes as in the default assignment of `MPI`. Note that we can always ensure this by creating a custom communicator. An important feature of this priority definition is the strictly increasing value of the key, i.e., the priority of a rank for a node cannot decrease. The idea behind this priority criterion is to ensure to assign a subgraph to the computation node that has as many edges as possible between the processes, ideally a clique. In Figure 4.1 a visualization on how the priority queue should work is presented.

The calculation of the permutation order can in principle be done by any process. In our design, we always choose the process with rank 0 (the root) to compute the reordering. The algorithm will iterate over all computation nodes, assigning the ranks in order given by the priority queue by repeatedly extracting the rank with the highest number of neighbor on the node and in the queue. In order to make sure that all processes are assigned exactly once, the root process keeps track of already assigned ranks. It stores an array of integers consisting of the new ranks. To be more specific, the new rank of process i is stored in position i of the array. The array containing the new ranks will be broadcast to all other processes, after termination of the computation. At the beginning of the algorithm the priority queue is empty and the root enqueues itself. After its assignment (since it was the only element in the queue) all of its neighbors are pushed into the priority queue, and the next suitable rank is extracted. This is done until all ranks have been assigned.

4.1.2 Pseudocode

The full pseudocode for the complete approach is depicted in Algorithm 3.

Algorithm 3: Centralized Greedy Algorithm

```
    Input: grid $g$ with dimension sizes $(d_1, \dots, d_n)$,
    stencil $s = (\mathbf{r_1}, \dots, \mathbf{r_n})$,
    number of nodes $N$,
    number of processes per node $p_{\text{node}}$,
    rank of root $rank_{\text{root}}$
    Result: Array with the new ranks $new\ ranks$
greedy central($g, s, N, p_{node}, rank_{root}$)
 1  $index \leftarrow 0$;
 2  //array containing the priority for each rank, initially zero
 3  $priority[i] \leftarrow 0$
 4  $pq \leftarrow$ priority queue;
 5  $pq$.enqueue($rank_{\text{root}}$);
 6  foreach  $Node$ do
 7      while  $Node\ not\ full$ do
 8          if $pq\ not\ empty$ then
 9              $best\ rank \leftarrow pq$.top();
10          else
11              $best\ rank \leftarrow$ next smallest unassigned rank;
12          end
13          if $best\ rank\ is\ unassigned$ then
14              foreach $neighbor \in N(best\ rank)$ do
15                  if $neighbor\ is\ unassigned \land neighbor \notin pq$ then
16                      //Since newly discovered rank, add one to priority
17                      $priority[neighbor] \leftarrow 1$
18                      $pq$.enqueue($neighbor$);
19                      foreach $neighbor\ of\ neighbor \in N(neighbor)$ do
20                          if $neighbor\ of\ neighbor \in pq$ then
21                              $priority[neighbor] \leftarrow priority[neighbor] + 1$
22                              $priority[neighbor\ of\ neighbor] \leftarrow$
                                  $priority[neighbor\ of\ neighbor] + 1$
23                          end
24                      end
25                  end
26              end
27              $new\ ranks[index] \leftarrow best\ rank$;
28              $index \leftarrow index + 1$;
29          end
30      end
31      $pq$.clear();
32      $pq$.push(next smallest unassigned rank);
33  end
```

4.1.3 Runtime Analysis

The complexity of Algorithm 3 depends on multiple parameters, i.e., the number of computational nodes N, the number of processes on each computation node p_{node}, the number of dimensions d and the size of the neighborhood k.

We need to know all the communication neighbors for all processes. For that purpose, we can use the stencil to calculate the coordinates of the communication neighbors. With this information, we can use `MPI_Cart_coords` routine to extract the rank of each neighbor. The extraction of the neighbors can be done in a preprocessing step in $\mathcal{O}(kPd)$ steps, where P is the total number of processes. This is because, we have to iterate over all processes P, the k-neighborhood of each process and for each neighbor extract the coordinates in all the dimensions d.

For an efficient look-up of the priority, we store the number of neighbors on the node and in the queue in an array, updating only when we encounter new ranks. Removing the best rank in the priority queue takes $\mathcal{O}(\log |pq|)$ steps, where $|pq|$ denotes the number of elements in the queue. After which we have to iterate over all k-neighbors of the best rank, inserting them into the queue and adjusting their priority. With an appropriate data structure, the cost of inserting the neighbors and adjusting the priority can be done in $\mathcal{O}(1)$ steps, see Brodal, Lagogiannis and Tarjan [6]. For each neighbor of the best rank, we again have to iterate over all k-neighbors, since newly enqueued ranks will change the priority of already enqueued ranks, resulting in $\mathcal{O}(k^2)$ steps for the priority adjustment. The priority is only valid for one computation node, so we have to reset it after the rank assignment of a node is complete. Every neighbor of the processes assigned to the node have been visited by the algorithm, and therefore have a different value in the array holding all priorities. For the next run, we need to reset these values, otherwise we introduce a bias for these ranks. By remembering which processes have been visited, we are able to reset the priority array in at most $\mathcal{O}(kp_{\text{node}})$ steps. This leads to a total theoretical runtime of

$$\mathcal{O}(kPd + N(p_{\text{node}}(\log(|pq|) + k^2) + p_{\text{node}}k)). \tag{4.1}$$

In the worst case, for any pair of processes u, v in the priority queue, there is no edge (u, v) in the Cartesian graph. This leads to each process having $(k - 1)$ neighbors in the queue, resulting after the node fill-up in $\mathcal{O}(kp_{\text{node}})$ processes in the priority queue. Using this we can write

$$\begin{gathered}\mathcal{O}(kPd + N(p_{\text{node}}(\log(kp_{\text{node}}) + k^2) + p_{\text{node}}k)) = \\ \mathcal{O}(kPd + Np_{\text{node}}\log(kp_{\text{node}}) + Np_{\text{node}}k^2 + Np_{\text{node}}k) = \\ \mathcal{O}(kPd + Np_{\text{node}}\log(kp_{\text{node}}) + Np_{\text{node}}k^2)\end{gathered} \tag{4.2}$$

Since the total number of processes is given by $P = Np_{\text{node}}$, we can write Equation (4.2) as

$$\mathcal{O}(kPd + P\log(kp_{\text{node}}) + Pk^2). \tag{4.3}$$

We implemented the algorithm in C++ and used routines in the standard library for the priority queue.[1] The implementation of these does not allow for insertion and increase key routines in $\mathcal{O}(1)$ steps, neither does it support finding an element in $\mathcal{O}(\log(|pq|))$ steps, forcing us to resort the queue after the priorities of the ranks have changed. This leads to a runtime of

$$\mathcal{O}(kPd + Pkp_{\text{node}} + Pk^2). \tag{4.4}$$

Since this approach did not perform well in the evaluation in Chapter 6 in terms of improvement in the amount of inter-node communication cost, we did not bother to introduce further improvements to the runtime.

4.2 Distributed Greedy Algorithm

The approach presented above is obviously not scalable. Hence, we will first motivate the need for an improvement, describe how to extend the greedy approach and proceed by giving a high level description in form of pseudocode.

4.2.1 Motivation

In order to improve on the fully centralized approach presented above in terms of scalability, we can find a set of processes that use the same greedy technique to fill-up the computation nodes individually. That is, the set should be of the size of the number of computation nodes, where each process is then responsible to fill-up a node. This would allow to fill-up the nodes in parallel. Since we want to avoid communication between the processes and overlap in the assignments, the processes in the set should be as far away as possible from each other in terms of stencil directions, i.e., we want to find a set of vertices in the Cartesian graph induced by the stencil communication on the Cartesian grid with maximal distance. For that purpose, we can repeatedly start a breadth-first search from an initial rank and always add the last discovered point to the starting set of the next round until the starting set has the size of the number of computation nodes. Again, we

[1]We used a normal std::vector and the functions std::make_heap, std::heap_push and std::heap_pop, see https://en.cppreference.com/

assumed that the ranks were assigned consecutively to the computation nodes and choose 0 as initial rank for the BFS. In the current implementation, all processes perform the BFS search in order to determine whether they are part of the initial set. Since the performance in terms of inter-node communication reduction of this algorithm was even worse than the centralized greedy approach and the input graph sizes are small, we did not parallelize the breadth-first search.[2]

After the starting set of processes is determined, each process in the set can use the technique presented above to fill-up a node. The results of the local fill-up is broadcast to all processes in order to make sure that each rank has been assigned exactly once. Since the rank assignment was performed without any communication between the processes, it might well be that some rank have been assigned to several nodes, while others were not assigned at all.

Given the complete new reordering, we can extract the ranks that have been assigned multiple times and the ranks that have not been assigned to any node. There are several strategies on how to assign the unassigned ranks to the computation nodes. If we were only interested in good partitions in terms of inter-node communication cost, we could assign each unassigned rank to the computation node which holds the maximum number of communication neighbors. If this node contains only ranks that have been assigned once, i.e. there are no duplicates, we would have to unassign such a rank, not decreasing the total number of unassigned nodes. We could repeat this process until all ranks have been assigned. Estimating the runtime for such an approach is quite difficult, but small practical experiments have shown that this method is too expansive, so we did not pursue it any further. Instead, we reduced the potential assignment space of the unassigned ranks to merely those ranks, which have been assigned multiple times. For each unassigned rank u, we compared it to all the duplicate ranks d and assign it to the node where the difference in inter-node communication is maximal. That is, for each unassigned rank u we count the number of off-node communication partners if it would be assigned to a process which is currently assigned to a duplicate rank d. We do the same for the process holding the duplicate rank d. Let c_u and c_d be the amount of off-node communication partners of the unassigned process u and the duplicate rank d then we want to maximize

$$f(c_u, c_d) = c_d - c_u. \tag{4.5}$$

[2]There are several techniques for parallelizing BFS, see [2, 23] to name just a few.

4.2.2 Pseudocode

The pseudocode for the distributed greedy approach is depicted in Algorithm 4, the adapted version of the greedy fill-up strategy can be found in Algorithm 6 and the routine to make sure every rank has been assigned in Algorithm 5.

Algorithm 4: Distributed Greedy Approach

```
  Input: grid $g$ with dimension sizes $(d_1, \dots, d_n)$,
  stencil $s = (\mathbf{r_1}, \dots, \mathbf{r_n})$,
  number of nodes $N$,
  number of processes per node $p_{\text{node}}$,
  Result: Array with new ranks new ranks
distributed-greedy-fill-up($g, s, N, p_{node}$)
1 //Check if calling process is in node set
2 starting set ← repeated breadth-first search($g, s, N$)
3 if my rank ∈ starting set then
4     //Fill up local node with calling process's rank as starter
5     local assignment ← local fill-up($g, s, p_{\text{node}}$, my rank)
6 end
7 //Broadcast result to everyone new ranks ← Broadcast(local assignment)
8 //Check whether all ranks have been assigned exactly once
9 global check-up (new ranks)
```

Algorithm 5: Global Check-Up

```
    Input: Array of ranks $ranks$ with potential duplicates and unassigned ranks
    Result: $ranks$ where every rank is assigned exactly once
global check-up($ranks$)
 1  //Extract duplicate or unassigned ranks
 2  $duplicate\ ranks \leftarrow$ duplicates in $ranks$
 3  $unassigned\ ranks \leftarrow$ unassigned ranks of $ranks$
 4  foreach $u \in unassigned\ ranks$ do
 5      //biggest gain of assigning $u$ to $d$
 6      $biggest\ gain \leftarrow$ min
 7      foreach $d \in duplicate\ ranks$ do
 8          //We can extract the Node ID from the indices of $d$
 9          foreach $index\ of\ d \in ranks$ do
10              //Count off-node neighbors for both ranks
11              $c_d \leftarrow$ off-node neighbors of $d$
12              $c_u \leftarrow$ off-node neighbors of $u$
13              if $c_d - c_u > biggest\ gain$ then
14                  $biggest\ gain \leftarrow c_d - c_u$
15                  $best\ index \leftarrow index$
16              end
17          end
18      end
19      //assign $u$ to the best position
20      $ranks[best\ index] \leftarrow u$
21      if $d$ no longer duplicate then
22          remove $d$ from $duplicates$
23      end
24  end
```

Algorithm 6: Node fill-up

```
    Input: grid $g$ with dimension sizes $(d_1, \ldots, d_n)$,
    stencil $s = (\mathbf{r_1}, \ldots, \mathbf{r_n})$,
    number of processes per node $p_{\text{node}}$,
    starting rank $rank_{start}$
    Result: Array local assignment with ranks assigned to node
local fill-up $(g, s, p_{\text{node}}, rank_{start})$
 1  $index \leftarrow 0$;
 2  //array containing the priority for each rank, initially zero for all $i$
 3  $priority[i] \leftarrow 0$
 4  $pq \leftarrow$ priority queue;
 5  $pq$.enqueue($rank_{start}$);
 6  while Node not full do
 7      if pq not empty then
 8          $best\ rank \leftarrow pq$.top();
 9      else
10          $best\ rank \leftarrow$ next smallest unassigned rank;
11      end
12      if best rank is unassigned then
13          foreach $neighbor \in N(best\ rank)$ do
14              if $neighbor \notin node \wedge neighbor \notin pq$ then
15                  //Since newly discovered rank, add one to priority
16                  $priority[neighbor] \leftarrow 1$
17                  $pq$.enqueue($neighbor$);
18                  foreach $neighbor\ of\ neighbor \in N(neighbor)$ do
19                      if $neighbor\ of\ neighbor \in pq$ then
20                          $priority[neighbor] \leftarrow priority[neighbor] + 1$
21                          $priority[neighbor\ of\ neighbor] \leftarrow$
                              $priority[neighbor\ of\ neighbor] + 1$
22                      end
23                  end
24              end
25          end
26          $local\ assignment[index] \leftarrow best\ rank$;
27          $index \leftarrow index + 1$;
28      end
29  end
```

4.2.3 Runtime Analysis

The runtime performance for this approach can be decomposed into three steps. First, the centralized breadth-first search takes $\mathcal{O}(n+m)$ steps for a single run and a graph with n vertices and m edges. Repeating this N amount of times with a Cartesian graph that has kP edges and P vertices leads to

$$\mathcal{O}(N(P+kP)) = \mathcal{O}(kNP) \tag{4.6}$$

amount of steps.

For each node we have to extract the communication neighbors the algorithm discovers, which is again no more than kp_{node}. Hence, filling the node can be done in

$$\mathcal{O}(kp_{\text{node}}d + p_{\text{node}}(\log(kp_{\text{node}}) + k^2)). \tag{4.7}$$

The global check-up has to iterate over all ranks P in order to assess the unassigned ranks and duplicates. Let n_u be the number of unassigned ranks and n_d the number of duplicate ranks. Algorithm 5 will iterate over all unassigned ranks and compare its neighborhood on the node to the neighborhood of the duplicate ranks, leading to

$$\mathcal{O}(P + n_u n_d k). \tag{4.8}$$

This leads to a theoretical total runtime[3] of

$$\mathcal{O}(kNP + kp_{\text{node}}d + p_{\text{node}}(\log(kp_{\text{node}}) + k^2) + n_u n_d k). \tag{4.9}$$

4.3 Summary

In this chapter we presented two greedy approaches, one fully centralized and one with parallel node fill-up. The main idea was to fill the computation nodes with groups of vertices that are maximally connected to one-another. Both approaches do not leverage the structure of the specific problem, i.e., the Cartesian graph and the isomorphic communication, but can be applied to any graph with arbitrary communication patterns. Their obliviousness to the structure makes them quite expansive in terms of theoretical runtime, since the have to look at every vertex and edge in the communication graph C. One can easily extend both approaches to weighted communication graphs or non-uniform node sizes, by weighting the priority with the edge weight and by storing a list of the different node sizes and fill-up accordingly. In the next chapter, we present a method that is no longer bounded by the number of vertices or edges in the communication graph.

[3]In our implementation $\mathcal{O}(kNP + kp_{\text{node}}d + p_{\text{node}}(kp_{\text{node}} + k^2) + n_u n_d k)$

Chapter 5

Hyperplane Algorithm

In this chapter, we present the hyperplane algorithm. The algorithm works recursively, by partitioning the grid using a suitable criterion. It assumes, like the other approaches taken in this thesis that each computational node has the same number of processes. We will first motivate the approach taken with this algorithm, followed by a high-level description and some pseudocode. We proceed by proving the validity of the algorithm and derive a theoretical runtime. Finally, we conclude with a short summary about the presented work in this chapter.

5.1 Motivation

A major drawback of the greedy approaches described in the last chapter, is the obliviousness to the Cartesian structure. The big advantage of the Cartesian graphs compared to unstructured graphs is the quick way of calculating the number of processes on the grid.[1] An alternative approach, that would get rid of the need to iterate overall processes is to calculate the partitions from the macroscopic point of view. To be more precise, the partitions are not found by looking at the vertices of the communication graph C, but instead by splitting the Cartesian grid. We can recursively partition the Cartesian graph into two parts, until each partition can be assigned to a computation node, i.e., has the size of the number of processes per node. Partitioning the Cartesian graph arbitrarily can induce significant inter-node communication cost. Since we want to minimize the inter-node communication, we should partition the grid in such a way that there are as few communication edges between the partitions as possible.

[1] $p = \prod_{i=1}^{d} d_i$.

Suppose the communication is only done along one direction given by some vector, then splitting the Cartesian graph repeatedly along that direction would induce zero inter-node communication cost, since there cannot be communication edges between the partitions. If communication happens along two directions, we would like to split the Cartesian graph along some direction parallel to the communication s.t. the hypersurface of the cut is minimal. The algorithm should aim to find a cutting hyperplane in each partitioning step, that is maximally parallel to the directions given by the stencil, while minimizing the hypercut surface. By keeping the Cartesian graph structure through the partitioning process, one can save substantial computation time, since calculating the partition sizes can be done in $\mathcal{O}(d)$ instead of $\mathcal{O}(|V|)$, where d is the number of dimensions and $|V|$ the number of vertices in the Cartesian graph C. The basic idea behind the hyperplane approach is to recursively partition the Cartesian grid g as parallel as possible to the overall communication into two Cartesian subgrids g' and g'' where both have the size of a multiple of the number of processes per node p_{node}. The subgrids g' and g'' are again partitioned until there are N subgrids of size p_{node}. This approach enables us to calculate a permutation for the processes without the need for communication between the processes.

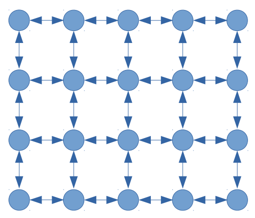
5.2 Algorithm

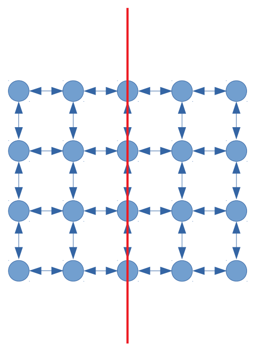
We will first give an verbal description of the algorithm, along with some explanations on the choice of dimension splitting criterion. After which, we give a high-level description in form of pseudocode.

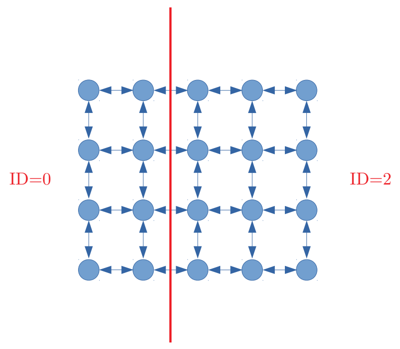
5.2.1 Formal Description

The concrete algorithm will work recursively. The final partition IDs correspond to the computation node IDs. At the beginning, there is only the total Cartesian graph, and we assign to it the partition ID 0. Each process knows the ID of the computation node it is assigned to. Using this information and the rank of the process on its assigned node, we can calculate its new coordinates in the grid after termination. To get rid of the need for communication, each process keeps track of the size and the position of the subgrid on the original grid. At each level of the recursion, the algorithm will perform the following steps.

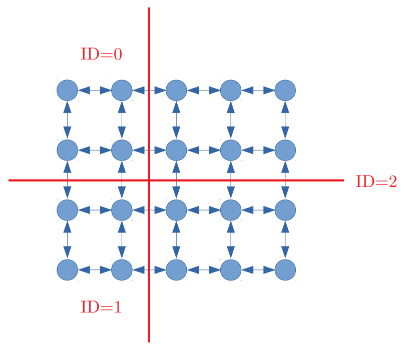
1. Find a permutation order of the dimension sizes s.t. the dimensions are ordered from the most to the least suitable, by some criterion. The $\cos(\alpha)$ as defined in Equation (5.1) between a relative coordinate vector $\mathbf{r_i} \in \mathscr{N} = \{\mathbf{r_1}, \ldots, \mathbf{r_k}\}$,


(a) Input

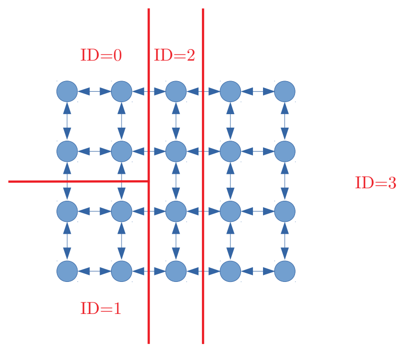

(b) Step 1



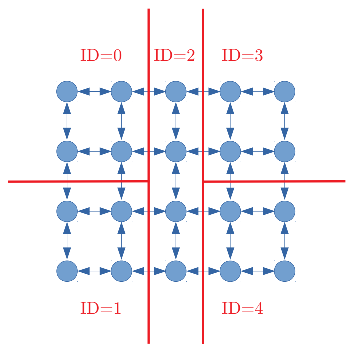
(c) Step 2



(d) Step 3



(e) Step 4



(f) Step 5

Figure 5.1: Motivating example for the hyperplane approach. (a): 5×4 input mesh with a five-point stencil, output 5 partitions of size 4. Because of the structure of the stencil, the Tuple (5.3) has the same value (2) for all the dimensions and the grid is only partitioned according to the dimension size. (b) and (c): Positioning of the first hyperplane s.t. both sides are a multiple of 4. (d): Recurse on both subgrids. On the left subgrid, we can immediately find a hyperplane s.t. both induced subgrids are of size 4 and we are done. On the right-hand side, the first suitable dimension cannot be split s.t. we can find subgrids with the size of multiple of 4, hence we try the next direction. (e): The left subgrid on the hyperplane is of size 4, and we are done. (f): Recurse on right subgrid and find the final hyperplane s.t. that all partitions are of size 4.

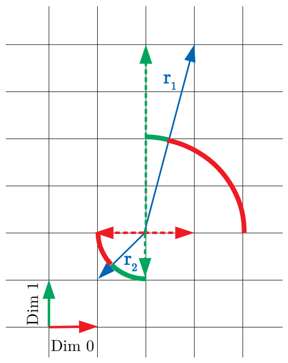


Figure 5.2: Motivating example to choose a dimension for the hyperplane. A 2D grid with a stencil (blue). Intuitively we want to split along dimension 0, since a split along dimension 1 would yield more cuts due to the component $\mathbf{r_1}$ in dimension 1. We can calculate $\cos(\alpha_{\mathbf{r_i},\mathbf{e_j}})$ between the relative coordinate vectors and the grid dimensions. One can see that the angles between the stencil directions and dimension 1 (green) are smaller than the angles between the stencil directions and dimension 0. Thus, we should first split along dimension 0.

given by a stencil and a grid directions $\mathbf{e_j} \in \mathscr{E} = \{\mathbf{e_1}, \dots, \mathbf{e_d}\}$ is proportional to the angle $\alpha_{\mathbf{r_i},\mathbf{e_j}}$ between $\mathbf{r_i}$ and $\mathbf{e_j}$ and thus a good quantifier of parallelism. If $\cos(\alpha)$ is zero for any $\mathbf{r_i}$, $\mathbf{e_j}$ pair, it means that $\mathbf{r_i}$ and $\mathbf{e_j}$ are parallel, if the absolute value is Equation (5.1) is one, then $\mathbf{r_i}$ and $\mathbf{e_j}$ are orthogonal to another.

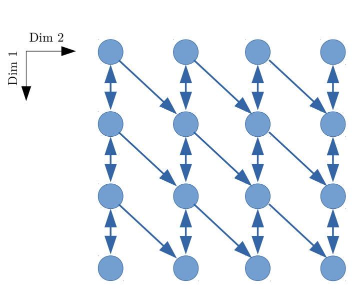
$$\cos(\alpha_{\mathbf{r_i}\mathbf{e_j}}) = \frac{\mathbf{r_i}\mathbf{e_j}}{||\mathbf{r_i}||\ ||\mathbf{e_j}||} \in [-1, 1] \tag{5.1}$$

To define a strictly monotonic indicator function, we can calculate the squared cosine between each neighbor and a dimension j, $1 \leq j \leq d$.

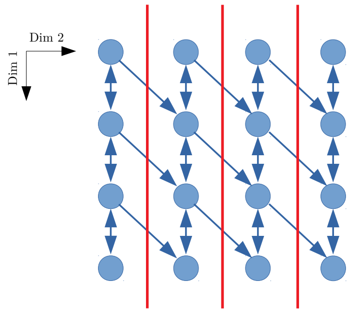
$$\sum_{i=1}^{k} \cos^2(\alpha_{\mathbf{r_i},\mathbf{e_j}}) \in [0, 1] \tag{5.2}$$

Doing so for all dimensions gives us

$$\left(\sum_{i=1}^{k} \cos^2(\alpha_{\mathbf{r_i},\mathbf{e_1}}), \dots, \sum_{i=1}^{k} \cos^2(\alpha_{\mathbf{r_i},\mathbf{e_d}}) \right). \tag{5.3}$$



(a) 4×4 grid with an asymmetrical stencil.



(b) Partitioning obtained by the hyperplane

Figure 5.3: Consider a 4×4 grid that should be mapped to four computation nodes, each having four cores and a stencil similar to the component stencil defined in Section 1.2.4 of Chapter 1, but with additional communication in a diagonal, see Figure 5.4. The Tuple (5.3) for that stencil has the following values $(2.5,\ 0.5)$. The input Cartesian graph is depicted in (a) and the output of the hyperplane partitioning in (b). Since the Tuple (5.3) has the minimal value in the second dimension, the hyperplane algorithm will split the grid along that dimension, i.e., find a hyperplane that is maximally parallel to the stencil communication.

The dimension with the minimum value in Equation (5.2), is the dimension that is orthogonal to the hyperplane, which is as parallel as possible to all relative coordinate vectors.[2] Note that by taking the square of the cosine, relative coordinate vectors $\mathbf{r_i}$ and grid direction vectors $\mathbf{e_j}$ that are strongly parallel $\cos(\alpha_{\mathbf{r_i},\mathbf{e_j}}) \approx 1$ will contribute more to the fit of a hyperplane than vectors that are weakly parallel. We can sort the Tuple 5.3 in ascending order to traverse the dimensions from most suitable to least suitable for a split. Note that each $\mathbf{e_{d_i}}$ has only one non-zero entry. This observation will improve the runtime for calculating Equation (5.2) for all dimensions.

2. Try to partition the grid into two parts, each being a positive multiple of the number of processes per node p_{node}. This is done by first positioning the

[2]This criterion only evaluates the dimensions according to their parallelism to the stencil directions. It is oblivious to the depth of the stencil communication. We could incorporate the depth of communication pattern, by not normalizing the inner product between $\mathbf{r_i}$ and $\mathbf{e_j}$. This would change Equation (5.2) to $\sum_{i=1}^{k} ||\mathbf{r_i}|| \cos^2(\alpha_{\mathbf{r_i},\mathbf{e_j}})$ for dimension j

hyperplane in the middle of the candidate dimension d', i.e., at $d'/2$. If the two partitions do not form a multiple of the number of processes per node p_{node}, then the hyperplane is shifted by one in the decreasing direction of the dimension. If no suitable split was found, i.e., we are at the last possible position for the hyperplane and cannot decrease the position any further, the next best dimension in the permutation order is tried. This step is repeated until a suitable split of a dimension d' is found. We prove in Section 5.2.3 that we are always able to find a split.

3. If the hyperplane split was found, we can calculate the number of partitions left on either side. This information is encoded in the IDs of the partitions. The left-hand side has partition ID id_{lhs}, which corresponds to the partition ID of the grid before the split, while the right-hand side receives partition ID $id_{lhs}+$ *number of partitions on the left-hand side*. Each process can then deduce on what side of the hyperplane it has to position itself, in order to be assigned to the correct computation node. The subgrid size and position on the grid is adjusted. If the number of remaining partitions on either side is greater than one that side will recurse and go back to step one.

4. If a partition has size of p_{node} then it enters the base-case. Each process can calculate individually its new coordinates, given the start and end points of the position of the subgrid on the overall grid. Suppose the partition of the calling process has the partition identity id and is on subgrid g_{sub} with dimension sizes $(p_1, \dots p_d)$ then we can calculate the new coordinates in the following manner.

   - First calculate the node rank R_{node} of the calling process with rank R on its computation node. Assuming that the processes were assigned consecutively to the computation nodes and that each computation node has the same amount of processes p_{node} assigned to it.

$$R_{node} = R - id * p_{node} \tag{5.4}$$

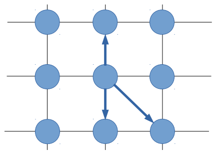


Figure 5.4: Example of an asymmetrical stencil.

- Then we can traverse the dimensions in order and assign to the coordinate vector holding the coordinates of the new rank the starting point of the position of the subgrid, i.e., we assign all processes to one vertex in the starting corner of the subgrid.
- In the next iteration over the dimensions, we assign to the coordinate in the current dimension i the remainder of the division between the R_{node} and the size of the dimension size p_i.

5.2.2 Pseudocode

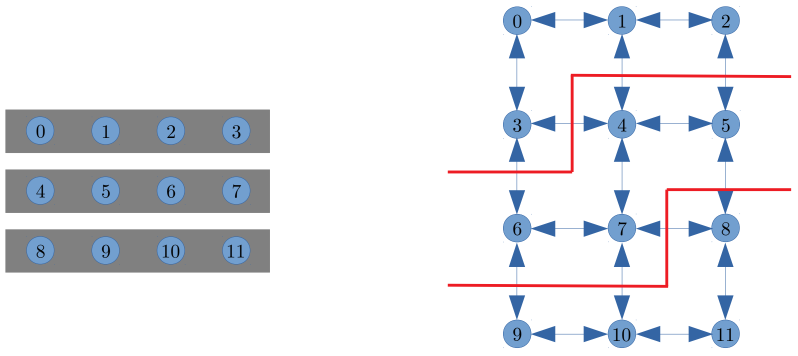
In this part, we provide the reader with a high-level notation of the hyperplane algorithm in form of pseudocode. The partitioning of the graph and the recursion steps are described in Algorithm 8, whereas the base-case new coordinate calculation is found in Algorithm 7.

Algorithm 7: Base-case

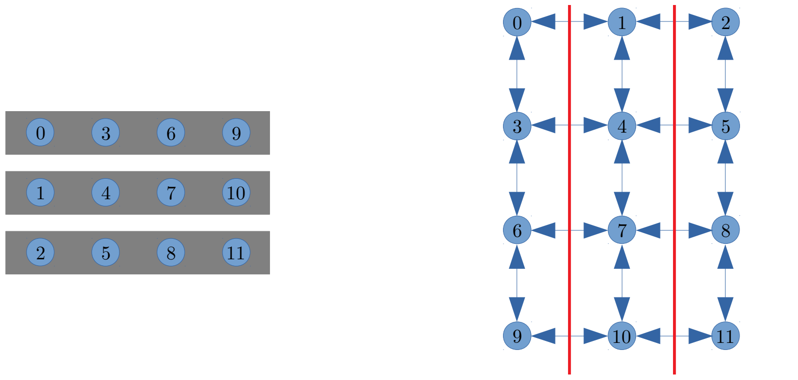
```
Input: subgrid $g$ with dimensions sizes $(p_1, \dots, p_d)$,
the subgrid's $position$ $(g_{sub,1}, \dots g_{sub,d})$ in the original grid,
the new rank $coordinates$,
number of processes per node $p_{node}$,
Partition identity $ID$,
rank $R$
Result: Coordinate vector of new rank
base-case($g, position, coordinates, p_{node}, ID$)
1  //calculate the node rank
2  $R_{node} \leftarrow R - ID \cdot p_{node}$;
3  //Assign all processes to the starting corner of the subgrid foreach $d' \leftarrow 1$ to
   $d$ do
4      $coordinates[d'] \leftarrow position[d']$;
5  end
6  $i \leftarrow 0$;
7  foreach $p'$ in dimension sizes do
8      $coordinates[i] \leftarrow coordinates[i] + R_{node} \mod p'$;
9      $i \leftarrow i + 1$;
10 end
```

Algorithm 8: Hyperplane Algorithm

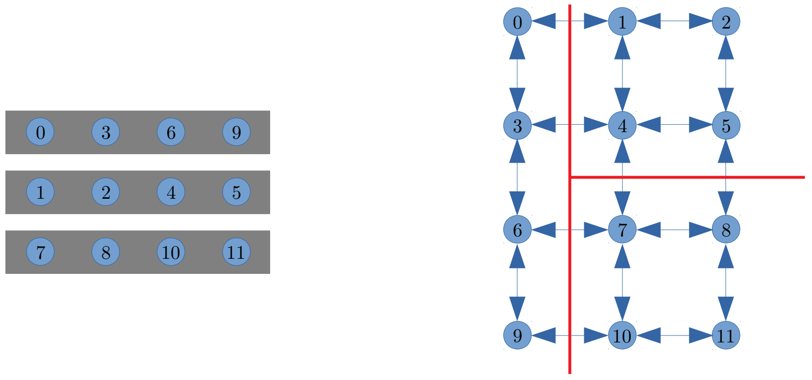
```
Input: grid $g$ with dimensions sizes $(p_1, \dots, p_d)$,
the grid's position in the original grid,
vector angles containing the sum over the neighbors of the squared $\cos(\alpha)$ per
dimension,
the new rank coordinates,
number of processes per node $p_{node}$,
Partition identity $ID = 0$
Result: Coordinate vector of new rank
hpp($g, position, angles, coordinates, p_{node}, ID$)
if $|g| = p_{node}$ then
    //calculate new coordinates of calling rank;
    base-case($g, position, coordinates, p_{node}, ID$)
    return;
end
//sort the dimensions based on the criterion defined in Equation (5.2)
//from best dimension to worst dimension
dimension permutation $\leftarrow$ sort(dimensions)
foreach $d'$ in dimension permutation do
    $h \leftarrow \frac{|g|}{d'}$;
    $a \leftarrow \left\lfloor \frac{d'}{2} \right\rfloor$;
    $b \leftarrow \left\lceil \frac{d'}{2} \right\rceil$;
    while $a \cdot h \mod p \neq 0 \land a > 1$ do
        $a \leftarrow a - 1$;
        $b \leftarrow b + 1$;
    end
    if $a \cdot h \mod p_{node} = 0$ then
        //remaining partitions in left-hand side
        $r \leftarrow \frac{a \cdot h}{p_{node}}$
        if $Node\ ID < ID + r$ then
            adapt positioning of subgrid;
            $d' \leftarrow a$;
            //recurse
            hpp($g, position, angles, coordinates, p_{node}, ID$);
        else
            adapt positioning of subgrid;
            $d' \leftarrow b$;
            //recurse
            hpp($g, position, angles, coordinates, p_{node}, ID + r$);
        end
        break;
    end
end
```



(a) Default



(b) Gropp's reordering



(c) Hyperplane reordering

Figure 5.5: Different reorderings for a 4×3 Cartesian graph with a five-point stencil to be mapped onto 3 computation nodes, each with 4 cores. On the left are the ranks of the processes on the cores (grey rectangles). The red lines on the right symbolize the partitions. (a): Default order of processes on the computation node, resulting in 16 total and one computation node with 8 inter-node communication edges, counting directed edges. (b): For illustration purposes we include Gropp's rearrangement here, having also 16 total and one node with 8 inter-node communication edges. (c): The hyperplane reordering would partition the Cartesian graph s.t. each node has 4 inter-node communication edges, resulting in a total of 12.

5.2.3 Validity and Runtime

The main part of the algorithm is to find a hyperplane that can split the grid g into two subgrids g' and g''. We will now show that it is always possible to find a hyperplane for a suitable mesh.

Theorem 5.2.1. *Let $C \in \mathbb{N}$ and $C \geq 2$. Let $p_{node} \coloneqq$ number of processes per node, with $p_{node} \in \mathbb{N}^+$. Let g be a d-dimensional grid, with dimensions sizes $D = (d_1, \dots, d_n)$ and $\forall d_i \in D : d_i \in \mathbb{N}^+$ and $|g| \coloneqq \prod_{i=1}^{d} d_i$ be the number of vertices of grid g. If $|g| = Cp_{node}$, then it is always possible to find a dimension d' that partitions g into two subgrids g_1 and g_2, s.t. $|g_1| = c'p_{node}$ and $|g_2| = c''p_{node}$ with $c', c'' \in \mathbb{N}^+$.*

Proof. The number of vertices is given by,

$$\prod_{i=1}^{d} d_i = Cp_{node}. \tag{5.5}$$

Let $F(x) = (f_{x1}, \dots, f_{xl})$ be all the l prime factors of $x \in \mathbb{N}$, i.e.,

$$x = \prod_{j=1}^{l} f_{xj}. \tag{5.6}$$

Writing every d_i in 5.5 as the product of its prime factors, we obtain

$$\prod_{i=1}^{d} \prod_{j=1}^{l_i} f_{d_i j} = Cp_{node}. \tag{5.7}$$

Note that the left-hand side of Equation (5.7) is also the product of the prime factors of C and p_{node}. That is, we can write the prime factorization of C and p_{node} as the product of prime factors corresponding to the dimension sizes.

$$C = F(C) = \prod_{\text{for some } f_{d_i j} \in F(\prod_{i=1}^{d} d_i)} f_{d_i j} \tag{5.8}$$

and

$$p_{node} = F(p_{node}) = \prod_{\text{for some } f_{d_i j} \in F(\prod_{i=1}^{d} d_i)} f_{d_i j}. \tag{5.9}$$

Since $F(\prod_{i=1}^{d} d_i) = F(C)F(p_{node})$, no $f_{d_i j}$ is used in both Equation (5.8) and Equation (5.9). Then there exists a $f_{d'j''}$ in $F(C)$ for which holds that $f_{d'j''} \geq 2$, which corresponds to a dimension d'. Write $f_{d'j''} = a + b$ with $a, b \in \mathbb{N}^+$, then we obtain for d'

$$d' = (a + b) \prod_{j \neq j''} f_{d'j} \tag{5.10}$$

enabling us to write Equation (5.5) as

$$\begin{aligned} (a + b) \prod_{j \neq j''} f_{d'j} \prod_{d_i \neq d'} d_i &= Cp_{node} \\ p_{node}(a + b) \prod_{d_C \in F(C) \setminus f_{d'j''}} f_C &= Cp_{node} \end{aligned} \tag{5.11}$$

where we can see that we can always split a dimension into two parts, resulting in two subgrids g_1 and g_2 with

$$|g_1| = p_{node} a \prod_{f_C \in F(C) \setminus f_{d'j''}} f_C \tag{5.12}$$

and

$$|g_2| = p_{node} b \prod_{f_C \in F(C) \setminus f_{d'j''}} f_C. \tag{5.13}$$

□

Secondly, we want to show that if there is dimension that is normal to every stencil communication direction then splitting this dimension will not induce any inter-node communication cost. For the proof, we introduce the following lemma stating that if all relative communication neighbor vectors $\mathbf{r_i} \in \mathscr{N}$ are normal to a dimension d_j then their j-th component is zero.

Lemma 5.2.2. *Let* $\mathbf{r}$ *and* $\mathbf{e_j}$ *be n-dimensional vectors and* $\mathbf{e_j}$ *have unit length with only one non-zero entry at position j. If* $\mathbf{r} \perp \mathbf{e_j}$*, then component j of* $\mathbf{r}$ *is zero.*

Proof. If $\mathbf{r} \perp \mathbf{e_j}$, then

$$\cos(\alpha) = \frac{\mathbf{r}\mathbf{e_j}}{||\mathbf{r}|| \; ||\mathbf{e_j}||} = 0. \tag{5.14}$$

Since $\mathbf{e_j}$ has unit length and only one non-zero element, the equation above is equivalent to

$$\cos(\alpha) = \frac{r_j}{||\mathbf{r_i}||} = 0 \tag{5.15}$$

where r_j is the j-th component of vector $\mathbf{r}$. One can easily see that for Equation (5.15) to be fulfilled, r_j must equal to zero. □

Theorem 5.2.3. *Let g be a d-dimensional grid with dimension sizes $D = (d_1, \dots, d_d)$. Let $\mathbf{e_i}$ be the unit vector indicating the direction of dimension d_i, with only one non-zero element at position i. Let $\mathscr{N} = \{\mathbf{r_1}, \dots, \mathbf{r_k}\}$ be the set of relative communication neighbor vectors. If there exists an $\mathbf{e_j}$ s.t. $\forall \mathbf{r_i} \in \mathscr{N} : \mathbf{r_i} \perp \mathbf{e_{d_j}}$ with $1 \leq j \leq d$ then a split of dimension d_j into $a, b \in \mathbf{N}^+$ with $d_j = a + b$ cannot induce any inter-node communication cost.*

Proof. Inter-node communication cost can only arise if there is a relative communication neighbor vector going from a subgrid to another.

A split of d_j into a and b will induce two subgrids g' with the same dimension sizes as the original grid g except for the j-th component $D = (\dots, d_j = a, \dots)$ and g'' with $D = (\dots, d_j = b, \dots)$. Let v' be a vertex in subgrid g' with coordinates $\mathbf{v'} = (v'_1, \dots, v'_d)$ and v'' be a vertex in subgrid g'' with coordinates $\mathbf{v''} = (v''_1, \dots, v''_d)$. Note that $\forall v' : v'_j \leq a$ and $\forall v'' : v''_j > a$. Inter-node communication is induced if there are two vertices v' and v'' s.t. $c(\mathbf{v'} - \mathbf{v''}) \in \mathscr{N}$ with $c \in \{-1, 1\}$. That is, there exists a pair v' in g' and v'' in g'' whose relative difference corresponds to a relative communication neighbor vector. The j-th component of $c(\mathbf{v'} - \mathbf{v''}) \neq \mathbf{0}$, since $\forall (v', v'') : v'_j \neq v''_j$. Hence, the vector $c(\mathbf{v'} - \mathbf{v''}) \notin \mathscr{N}$ since $\forall \mathbf{r} \in \mathscr{N} : r_j = 0$. □

We will now prove that this algorithm will yield unique rank coordinates to every rank in the original grid g.

Theorem 5.2.4. *Let g be a d-dimensional grid with dimension sizes $(d_1, \dots, d_d)$. Let g be partitioned into N subgrids, as described in Algorithm 8. Let the subgrid g_i, with $0 \leq i \leq N - 1$ have dimension sizes $(d_{i,1}, \dots, d_{i,d})$ and starting points for each dimension $(g_{i,1}, \dots, g_{i,d})$ in the original grid g. For each subgrids g_i holds that $\prod_{j=1}^{d} d_{i,j} = p_{node}$. Then when Algorithm 8 completes, each process with rank R in g will have a unique new coordinate vector $(R_1, \dots, R_d)$, describing the position of the new rank of R.*

Proof. First, observe that the subgrids g_i will always be disjoint, due to the way the split is defined. A split of a grid g around a dimension k, by splitting the dimension size $p_k = a + b$ will induce two subgrids g' and g''. All vertices v' in g' have a value smaller or equal than a in the k-th component $v'_k \leq a$, whereas all vertices v'' in g'' have a value strictly bigger than a in the k-th component $v''_k > a$. This argument holds for each split resulting in

$$\bigcup_{i=0}^{N-1} g_i = g \quad \text{and} \quad \bigcap_{i=0}^{N-1} g_i = \emptyset. \tag{5.16}$$

When entering the base case, all the processes in a subgrid g_i will be assigned to the same vertex at position $(g_{i,1}, \ldots, g_{i,d})$ as described in Algorithm 7. That means there is one vertex in the subgrid with a load of p_{node}, while there $p_{node} - 1$ vertices with load 0. In the first step of Algorithm 7 in dimension 1, all processes will update their coordinate to the remainder of the division between their rank on the node R_{node} and $d_{i,1}$ (Line 8 in Algorithm 7). Since all processes have a unique node rank between $0 \leq R_{node} \leq p_{node} - 1$ and $p_{node} = d_{i,1} \prod_{j=2}^{d} d_{i,j}$, each vertex v with coordinates $(g_{i,1} \leq v_1 \leq g_{i,1} + d_{i,1}, \ldots, g_{i,d})$ has a load of $\frac{p_{node}}{d_{i,1}}$, whereas there are $p_{node} - d_{i,1}$ vertices with load 0. Note that taking the remainder of the division ensures that the processes will be assigned to vertices within the subgrid. In general after m steps, there will be $\prod_{j=1}^{m} d_{i,j}$ vertices with load $\frac{p_{node}}{\prod_{j=1}^{m} d_{ij}}$ and $p_{node} - \prod_{j=1}^{m} d_{i,j}$ with load 0. After d steps all vertices in the subgrid will have a load of 1, i.e., each process will be assigned to a unique position in the subgrid and since all subgrids are disjoint and all processes in the overall grid g will be assigned to a unique rank. □

One can easily see that Algorithm 8 always terminates.

Theorem 5.2.5. *Let g be a d-dimensional grid with dimension sizes $(d_1, \ldots, d_d)$. Assume that there are N computation nodes and let p_{node} be the number of processes per node. Let $\prod_{i=1}^{d} d_i = N p_{node}$ then Algorithm 8 will produce N partitions with size p_{node} and always terminate.*

Proof. In each call, Algorithm 8 will either enter the base-case if the size of the input grid is p_{node} and thus terminate or it will partition g into two subgrids g' and g'' with new dimension sizes $(d'_1, \ldots, d'_d)$ and $(d''_1, \ldots, d''_d)$, where at least one d_i is smaller than in the original grid. For both subgrids holds that $\prod_{i=1}^{d} d'_i = c' p_{node}$ and $\prod_{i=1}^{d} d''_i = c'' p_{node}$ for some $c', c'' \in \mathbb{N}^+$. The subgrids always fulfill the termination criteria, namely that their size is a multiple of p_{node}. Since the input grid size can only decrease, the algorithm terminates after a finite amount of steps. □

The runtime analysis is a bit more complicated, since it is a recursive problem and the input size for each level can vary.

At first, we can exploit the fact that the subgrid dimension directions on each recursive level are parallel to those of the original grid. Hence, we can compute the angle between each $\mathbf{r_i} \in \mathscr{N}$ and $\mathbf{e_j} \in \mathscr{E}$ in a preprocessing step in $\mathcal{O}(kd)$ steps, where $k = |\mathscr{N}|$ and d is the number of dimensions.

Theorem 5.2.6. *Given a set of unit vectors $\mathscr{E} = (\mathbf{e_1}, \ldots, \mathbf{e_n})$ with only one non-zero element and a set of relative neighbor vectors $\mathscr{N} = (\mathbf{r_1}, \ldots, \mathbf{r_k})$, then it is possible to* *calculate the sum of all squared* $\cos(\alpha)$ *as defined in 5.2, where* α *is the angle between some* $\mathbf{r_i} \in \mathscr{N}$ *and some* $e_j \in \mathscr{E}$ *in* $\mathcal{O}(kd)$ *steps.*

Proof. We take each of the k relative vector $\mathbf{r_i} \in \mathscr{N}$, calculate its length $||\mathbf{r_i}||$ in $\mathcal{O}(d)$ steps. We can now use the observation made above that all $\mathbf{e_j} \in \mathscr{E}$ have exactly one non-zero element in the j-th component and unit length. Denote $r_{i,j}$ the j-th component of $\mathbf{r_i}$ then 5.1 becomes

$$\cos(\alpha) = \frac{\mathbf{r_i}\mathbf{e_j}}{||\mathbf{r_i}||||\mathbf{e_j}||} = \frac{r_{i,j}}{||\mathbf{r_i}||}. \tag{5.17}$$

Now each computation of α can be done in $\mathcal{O}(1)$ steps, as does the computation of the squared value and the addition to already calculated values. As this is done over all dimensions, in total this gives us $\mathcal{O}(dk)$ steps. □

Let us now look at the amount of work done in each recursive call, excluding the amount of work done in the consecutive levels.

The first step is to find a permutation of the order of dimension sizes. Since the calculation of the squared $\cos(\alpha)$ values was done as a preprocessing step, we only need to sort the dimensions sizes accordingly. Sorting can be done in $\mathcal{O}(d\log(d))$ steps, where d is the number of dimensions. Since we only change the one dimension size in each recursive call, we can presort the dimension sizes according to Equation (5.2) and then only update the position of the changed the dimension size d'. This approach allows us to sort the dimension in the recursive steps in $\mathcal{O}(d)$ steps.

In the second step, we try to find a suitable position for the hyperplane.

Theorem 5.2.7. *Let* g *be a* d*-dimensional grid with dimensions sizes* $D = (d_1, \dots, d_d)$ *with* $\forall d_i \in D : d_i \in \mathbb{N}^+$*. Let* $|g| = \prod_{i=1}^{d} d_i = Cp$ *for some* $C, p \in \mathbb{N}^+$*. Then it is possible to find a dimension* d' *s.t. a split of* d' *fulfills the criteria defined in Theorem 5.2.1 in* $\mathcal{O}(\sum_{i=1}^{d} d_i)$ *steps.*

Proof. Denote d' as the candidate dimension. Then we split d' into two parts, writing $d' = a + b$ with $a = \left\lfloor \frac{d'}{2} \right\rfloor$ and $b = \left\lceil \frac{d'}{2} \right\rceil$. For symmetry reasons, we only need to look at half of the possible positions for the hyperplane. Note that we move the position of the split in d' at most $\frac{d'}{2}$ times. In the worst case, we have to look at all dimensions. Resulting in $\sum_{i=1}^{d} \frac{d_i}{2}$ steps. □

In the best case scenario the first dimension d' in the permutation order of the dimension sizes will find a suitable split at $\frac{d'}{2}$. This would give us $\mathcal{O}(d\log(d))$ amount of work in one recursive call, since we only need to sort the dimension sizes and all other operations are of order $\mathcal{O}(1)$. In the worst case, we would need to iterate over all possible dimensions, resulting in a runtime of $\mathcal{O}(d\log(d) + \sum_{i=1}^{d} d_i)$.

As mentioned before, the amount of partitions left on either side can be calculated in $\mathcal{O}(1)$ steps. After finding the correct split, the subgrid sizes and the number of processes per subgrid can be calculated in $\mathcal{O}(1)$ time, since we can use $(a + b)\prod_{d \in D \setminus d'} d$. For the position of the subgrids in the grid we need d points. We just have to adapt one point in dimension d' for the left-hand and the right-hand side of grid respectively, which can be done in $\mathcal{O}(1)$ steps.

When the algorithm enters the base case, we can calculate the coordinates of the calling rank in the overall grid in $\mathcal{O}(d)$ steps.

Theorem 5.2.8. *The runtime of Algorithm 7 is $\mathcal{O}(d)$.*

Proof. Calculating the node rank R_{node} can be done in constant time. The two loops are over the dimensions and inside of each loop a constant number of steps is done. Resulting in $\mathcal{O}(d)$ steps in total. □

If we take a look at the overall runtime, we see that the inner work on each level of the recursion tree affects the number of recursive calls. If we find a hyperplane that splits each dimension d' in middle, i.e., at $\frac{d'}{2}$ then there are $(\log_2(N) - 1)$ calls, where N is the number of computation nodes. In the worst case on the other hand, at each level of the recursion tree we can only partition the dimension size d' into $a = 1$ and $b = d' - 1$ then we would need a total of $\mathcal{O}(N)$ calls.

Before giving an estimation for the total amount of recursive calls, observe that the worst case of only being able to find a partition of size p_{node} can arise in just two cases.

Theorem 5.2.9. *Let g be a d-dimensional grid with dimension sizes $D = (d_1, \ldots, d_d)$. Let $|g| = \prod_{i=1}^{d} d_i$ be the size of a grid. Let $\prod_{i=1}^{d} d_i = Cp_{node}$, where p_{node} is the amount of processes per computation node and $C \in \mathbb{N}$ and $C \geq 2$. Then Algorithm 8 will find a split of g into two subgrids g' and g'', where either $|g'| = p_{node}$ and/or $|g''| = p_{node}$ if and only if C is either 2 or 3.*

Proof. Consider the product of the prime factors of the candidate dimension size Equation (5.7). For the purpose of this proof it suffices to look at the prime factors of C.

$$\prod_{j=1}^{l_C} f_{Cj} = C. \tag{5.18}$$

Assume that $l_C > 1$, that is C has more than one prime factor. Then a split of any prime factor $f_{Cj'} = a + b$ would result into two partitions both greater than p_{node}, since

$$\begin{gathered}
p_{node}(a+b) \prod_{f_{Cj} \neq f_{Cj'}} f_{Cj} = C p_{node} = \\
\left(a \prod_{f_{Cj} \neq f_{Cj'}} f_{Cj} + b \prod_{f_{Cj} \neq f_{Cj'}} f_{Cj} \right) p_{node} \\
\underbrace{a p_{node} \prod_{f_{Cj} \neq f_{Cj'}} f_{Cj}}_{|g'| \geq p_{node}} + \underbrace{b p_{node} \prod_{[f_{Cj} \neq f_{Cj'}} f_{Cj}}_{|g''| \geq p_{node}} \\
\text{since} \prod_{f_{Cj} \neq f_{Cj'}} f_{Cj} > 1.
\end{gathered} \tag{5.19}$$

Thus, $f_{Cj} = C \geq 2$. Since the prime factorization is equal to that of Cp_{node}, C must also be the prime factor of some dimension size d'. Lets us write the dimension size d' as the product of its prime factors.

$$C \prod_{f_{d'} \in F(d') \setminus C} f_{d'} = d' \tag{5.20}$$

In line 11, Algorithm 8 will divide d' into $a = \left\lfloor \frac{d'}{2} \right\rfloor$ and $b = \left\lceil \frac{d'}{2} \right\rceil$. Since

$$p_{node} = \frac{\prod_{i=1}^{d} d_i}{C} \tag{5.21}$$

the only factor that is allowed to be modified in 5.20 is C, since any other modification would change p_{node}. If d' is even then

$$\begin{aligned}
a &= \left\lfloor \frac{C}{2} \prod_{f_{d'} \in F(d') \setminus \{f'\}} f_{d'} \right\rfloor = \frac{C}{2} \prod_{f_{d'} \in F(d') \setminus \{f'\}} f_{d'} \\
b &= \left\lceil \frac{C}{2} \prod_{f_{d'} \in F(d') \setminus \{f'\}} f_{d'} \right\rceil = \frac{C}{2} \prod_{f_{d'} \in F(d') \setminus \{f'\}} f_{d'}.
\end{aligned} \tag{5.22}$$

and clearly for a partition to have size p_{node}, $C = 2$.

If d' is odd, then

$$\begin{aligned} a &= \left\lfloor \frac{C}{2} \prod_{f_{d'} \in F(d') \setminus \{f'\}} f_{d'} \right\rfloor = \frac{(C \prod_{f_{d'} \in F(d') \setminus \{f'\}} f_{d'}) - 1}{2} \\ b &= \left\lceil \frac{C}{2} \prod_{f_{d'} \in F(d') \setminus \{f'\}} f_{d'} \right\rceil = \frac{(C \prod_{f_{d'} \in F(d') \setminus \{f'\}} f_{d'}) + 1}{2}. \end{aligned} \tag{5.23}$$

There are two values for C to obtain a partition of size p_{node}. Either $C = 2 \implies \left\lfloor \frac{C}{2} = 1 \right\rfloor$ or $C = 3 \implies \left\lfloor \frac{C}{2} = 1 \right\rfloor$. □

This gives us hope that in the worst case the depth of the recursion tree is still logarithmic in the number of computation nodes. In fact, each hyperplane will produce two subgrids where the ratio of the sizes is bounded.

Theorem 5.2.10. *Let g be a d-dimensional grid with dimension sizes $D = (d_1, \dots, d_d)$. Let $\prod_{i=1}^{d} d_i = Cp_{node}$, where C, $p_{node} \in \mathbb{N}^+$ and $C \geq 2$. Then Algorithm 8 will always partition g into two subgrids g' and g'' s.t. the $\frac{1}{2} \leq \frac{|g'|}{|g''|} \leq 1$.*

Proof. We will again use the fact that the prime factorization of $\prod_{i=1}^{d} d_i$ and Cp_{node} are equal. Therefore, we can split the l_i prime factors of each dimension size $d_i \in D$ into m prime factors d_i contributing to C and $l_i - m$ prime factors contributing to p_{node}.

$$d_i = \underbrace{d_{i,1}, \dots, d_{i,m}}_{\text{Contributing to } C} \underbrace{d_{i,m+1}, \dots, d_{i,l_i}}_{\text{Contributing to } d_{node}} \qquad m \leq l_i. \tag{5.24}$$

Let the prime factors $d_{i,m}$ contributing to C be ordered, i.e., $d_{i,1} \leq \cdots \leq d_{i,m}$. Let the candidate dimension size d' have prime factors contributing to C, i.e., $m > 0$. Note that if $m = 0$, Algorithm 8 will iterate over the candidate dimension size d' without finding a suitable split. Then the algorithm will surely find a suitable split at d'_1 with $\left\lfloor \frac{d'_1}{2} \right\rfloor$ and $\left\lceil \frac{d'_1}{2} \right\rceil$, since the prime factors contributing to p_{node} remain unchanged. If $d'_1 = 2$, the resulting split will yield two partitions of exactly the same size.

$$\begin{aligned} |g'| &= \frac{d'_1}{2} \prod_{j=2}^{m} d'_j \prod_{d_i \neq d'} d_i \\ |g''| &= \frac{d'_1}{2} \prod_{j=2}^{m} d'_j \prod_{d_i \neq d'} d_i \\ \implies \frac{|g'|}{|g''|} &= 1. \end{aligned} \tag{5.25}$$

If $d'_1 \geq 3$, then a split around it will yield two partitions with a bigger difference than a split of any other prime factor d'_k contributing to C with $2 \leq k \leq m$. Note that in this case $\left\lfloor \frac{d'_1}{2} \right\rfloor = \frac{d'_1 - 1}{2}$ and $\left\lceil \frac{d'_1}{2} \right\rceil = \frac{d'_1 + 1}{2}$.

$$\begin{gathered} \frac{\frac{d'_1-1}{2} \prod_{j=2}^{m} d'_j \prod_{d_i \neq d'} d_i}{\frac{d'_1+1}{2} \prod_{j=2}^{m} d'_j \prod_{d_i \neq d'} d_i} \leq \frac{\frac{d'_k-1}{2} \prod_{j \neq k}^{m} d'_j \prod_{d_i \neq d'} d_i}{\frac{d'_k+1}{2} \prod_{j \neq k}^{m} d'_j \prod_{d_i \neq d'} d_i} \\ = \\ \frac{d'_1 - 1}{d'_1 + 1} \leq \frac{d'_k - 1}{d'_k + 1}. \end{gathered} \tag{5.26}$$

This always holds, since this is a strictly monotonic increasing function that converges to 1 for growing d'_k values. To see this suppose $x, y \geq 0$

$$\begin{aligned} \frac{x-1}{x+1} &\leq \frac{y-1}{y+1} \\ (x-1)(y+1) &\leq (y-1)(x+1) \\ xy + x - y - 1 &\leq xy + y - x - 1 \\ x &\leq y. \end{aligned} \tag{5.27}$$

Again this will satisfy the splitting criteria, since prime factors contributing to p_{node} remain unchanged. Note that if the Algorithm 8 first positions the hyperplane at $\left\lfloor \frac{d'}{2} \right\rfloor$ it will eventually find a suitable split, latest at $\left\lfloor \frac{d'_1}{2} \right\rfloor \prod_{j=2}^{m} d'_j$.

$$\begin{aligned}\left\lfloor \frac{d'_1}{2} \right\rfloor \prod_{j=1}^{m} d'_j &\leq \left\lfloor \frac{d'}{2} \right\rfloor \\ \frac{d'_1 - 1}{2} \prod_{j=1}^{m} d'_j &\leq \frac{d' - 1}{2} \\ d' - \prod_{j=2}^{m} d'_j &\leq d' - 1 \\ -\prod_{j=2}^{m} d'_j &\leq -1\end{aligned} \tag{5.28}$$

We can bound the ratio of the two grid sizes $|g'|$ and $|g''|$ from below, since $d'_1 \geq 3$.

$$\frac{|g'|}{|g''|} = \frac{\frac{d'_1-1}{2} \prod_{j=2}^{m} d'_j \prod_{d_i \neq d'} d_i}{\frac{d'_1+1}{2} \prod_{j=2}^{m} d'_j \prod_{d_i \neq d'} d_i} \geq \frac{d'_1 - 1}{d'_1 + 1} \geq \frac{3-1}{3+1} \geq \frac{1}{2}. \tag{5.29}$$

□

Theorem 5.2.10 shows that the input grid always decreases at least by a factor of $\frac{2}{3}$, resulting in a recursion tree depth of maximal $\log_{\frac{3}{2}}(N)$ and that a candidate dimension d' is only iterated through, if all the prime factors of d' contribute to p_{node}.

At each level in the recursion tree, a worst-case of $\mathcal{O}(d + \sum_{i=1}^{d} d_i)$ is performed resulting in a total, theoretical runtime of

$$\mathcal{O}(dk + d\log(d) + \log(N)(d + \sum_{i=1}^{d} d_i)). \tag{5.30}$$

5.3 Summary

In this chapter, we introduced the hyperplane algorithm, which recursively splits the input Cartesian graph C into two subgraphs, while preserving the Cartesian structure throughout the recursion. We were able to show that finding these splits is always possible if the required assumptions are fulfilled, i.e., each computation node has the same number of processes per node p_{node}. Note that the algorithm could be extended to handle cases with inhomogeneous node sizes. We will give a first, intuitive idea in Chapter 7. Further, we could provide an upper-bound for the theoretical runtime, which is no longer dependent on the number of vertices and edges of the original Cartesian graph C. Another advantage of this algorithm is that the processes can calculate their new coordinates independently of another, without the need for communication. This results in very good scaling behavior and computational load balance compared to the two greedy approaches, that needed several broadcasting operations and the computation load was not well balanced, if at all. In Chapter 6 we will investigate the experimental runtime of the algorithm. More specifically, we look at the depth of the recursion tree and at the number of times the hyperplane must be shifted in order to find a suitable split.

Chapter 6

Experimental Evaluation

This chapter deals with the experimental evaluation of the presented algorithms. We will commence by defining the experimental setup in which we give information about the used hard- and software and the generated instances. We proceed with the experimental evaluation of the theoretical runtime of the hyperplane algorithm, in which we look at the recursion tree depth and the number of hyperplane shifts. After which, we continue by directly comparing the number of inter-node communication for different reordering schemes, including VieM. We then proceed by examining the effect of reduction in inter-node communication in terms of communication time needed for the `MPI_Neighbor_alltoall` routine. We conclude by comparing the instantation time needed for the Cartesian communicators with and without the presented reordering schemes.

6.1 Experimental Setup

The following section aims to clarify the setup of the experiments in terms of the used hard- and software, the grid setups and the stencils that were used to induce the Cartesian graphs. We will give information about the framework of the code and how one could use it.

6.1.1 Hardware and Software

All the experiments have been done on a system at the TU Wien called Hydra, an Intel Skylake-OmniPath with 36 computation nodes, each with two Intel Xeon Gold 6130 CPUs with a clock speed of 2.10 GHz, i.e., each node has 32 cores. Each node has 94 GB of main memory. We used Open Mpi 4.0.1. We implemented Gropp's

`Nodecart` routine as described in Algorithm 2 in Chapter 3, the two greedy approaches in Chapter 4 and the hyperplane algorithm described in Chapter 5. The code is written in C++ and compiled using the gcc 8.3.0 with full optimization flags (-O3).

Implementation Details

We implemented wrappers around the Cartesian instantiation routines. To be more precise, `Nodecart` and the algorithms developed in this thesis first calculate the new ranks, create a new communicator with the new ranks, which is then passed to the `MPI_Cart_create` routine to instantiate the new Cartesian communicator. The algorithms were designed to get a Cartesian communicator as an input, from which they are able to obtain all necessary information like the number of dimensions, the dimension sizes, the periodicity and the calling ranks coordinates.[1] The stencils can be passed as an integer list of size $k \cdot d$, where k is the number of neighbors and d the number of dimensions. The list holds the offset in each dimension for every communication neighbor, i.e., the relative distance vector describing the position of the first communication neighbor is given by the first d entries, the second communication neighbor by the second d entries and so on.

Listing 6.1: Function signature for the wrappers of the two greedy approaches and the hyperplane algorithm.

```
void MPIX_Cart_greedy_central(MPI_Comm cart_comm,
        const int stencil[],
        const int n_neighbors,
        MPI_Comm * greedy_central_comm);

void MPIX_Cart_greedy_dist(MPI_Comm cart_comm,
        const int stencil[],
        const int n_neighbors,
        MPI_Comm * greedy_dist_comm);

void MPIX_Cart_hyperplane(MPI_Comm cart_comm,
        const int stencil[],
        const int n_neighbors,
        MPI_Comm * hyperplane_comm);
```

The algorithms presented in this thesis expect the ranks to be assigned consecutively on the computation nodes and in order to ensure this, we included a routine that

[1]This information is cached in Cartesian communicator.

re-sorts the ranks accordingly. This is done by creating a communicator consisting of one process per computation node and communicators consisting of all processes on a node, similarly as it is done in the `Nodecart` approach described in Chapter 3. Given the two communicators and the fact that each node has the same number of processes, each process can calculate its new rank, ensuring that the processes lie consecutively on the nodes. If not mentioned otherwise, all the experiments have been done with an initial default rank assignment of `MPI`.

6.1.2 Grids

We used different of number of dimensions, nodes and processes per node to generate a variety of grid configurations. All experiments were done with at least 6 and up to 36 nodes with a step size of 3, for each node configuration we varied the number of processes on the node from 8 to 32 with a step size of 4 and choose the number of dimensions to be the number of prime factors of the product between the number of nodes and the number of processes on a node. For example, if we use 8 nodes and each node has 12 processes then the total number of processes is 96 which has 6 prime factors $(2, 2, 2, 2, 2, 3)$, so we let the number of dimensions of the grid range from 2 to 6. The dimension sizes were created with the `MPI_Dims_create` routine. We used solely non-periodic grids throughout the experiments.

6.1.3 Stencils

We give a short overview of the stencils used for the experiments and why we choose them. The stencil definitions can be found in Chapter 1 in Section 1.2.4.

We experimented with a variety of stencils, each giving some rank reordering strategies different advantages. We used the general five- and nine-point stencil, since they are used in a lot of benchmarks, including in Gropp's [14]. We designed the component stencil, which generates d_d components, where d_d is the grid size in the last dimension. The hope is that this stencil favours the hyperplane algorithm, since in each recursion level, it will try to find a split along the last dimension, as there is no communication alongside of it. Another stencil we used for benchmarking is the *diagonal* stencil. This stencil has no communication that is parallel to the grid basis. The hyperplane algorithm cannot partition the Cartesian graph parallel to any communication direction. This communication pattern is better suited for the greedy approaches. We included the *Crank-Nicolson* stencil in our benchmarks, which is an often used scheme in finite difference for solving partial differential equations, see Crank and Nicolson [9]. Finally, we also experimented with two stencils that have multiple hops to test the effect of *hops* on the algorithms. In our case, we performed tests with 3 hops on the general five-point stencil in the first and the last dimension, i.e., in either the first or the last dimension a process has 3 communication neighbors per direction instead of 1, see Figure 1.3b. This strongly favors partitioning along the dimension with the hops and the hope is that the hyperplane algorithms finds goods partitions.

6.2 Hyperplane Parameters

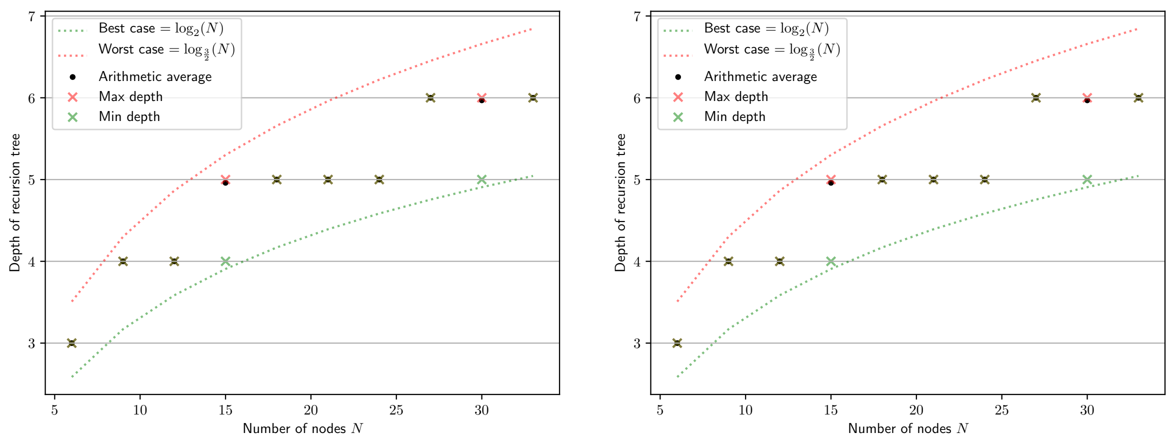
In order to visualize the dependency on the parameters for the theoretical runtime of the hyperplane algorithm defined in Equation (5.30) and see the practical runtime behaviour, we can count the amount of recursive calls and the amount of shifts of the hyperplane for each calling process and different stencil forms. We first look at the depth of the recursion tree and continue to investigate the maximal number of shifts in practice.

6.2.1 Depth of Recursion Tree

We have shown in Chapter 5, Theorem 5.2.10 that the worst case depth of the recursion tree is $\log_{\frac{3}{2}}(N)$, where N is the number of computation nodes, while the best case recursion tree depth is $\log_2(N)$, i.e., if the hyperplane algorithms always finds subgrids of the same size. We are interested to see in how close we are to the best case scenario in practice. We extracted the maximal number of recursive calls over all processes for all generated instances.

In Figure 6.1, we plot the maximal, minimal and average maximal depth over all instances of the recursion tree per number of nodes N for the general five- and nine-point stencil, in Figure 6.2, for the component and the Crank-Nicolson stencil, in Figure 6.3, for the general five-point stencil with 3 hops in the first and last dimension and in Figure 6.4 for the diagonal stencil.

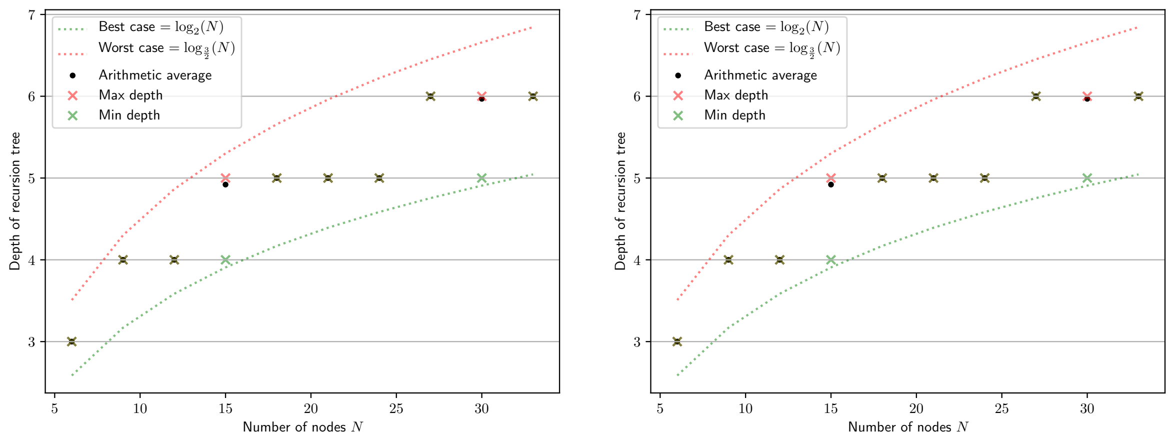
As one can see, on average we are very close to the rounded up best-case scenario. This indicates that the size ratio of the hyperplane induced subgrids is well balanced.



(a) General five-point stencil.

(b) General nine-point stencil.

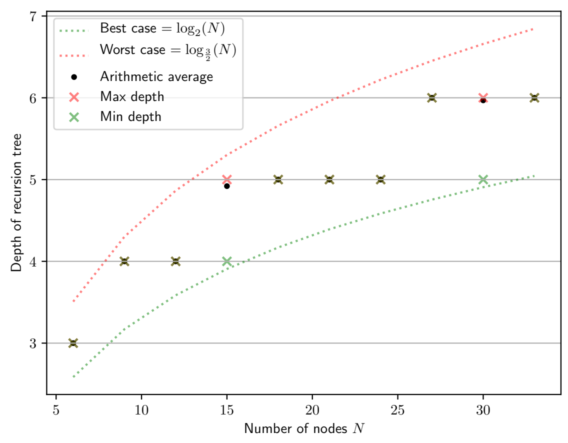
Figure 6.1: Maximal number of recursive calls for different instances per node for the general five-point stencil (a) and the general nine-point stencil (b).



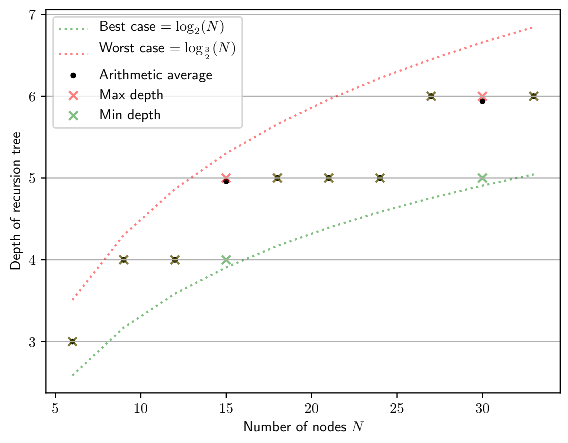
(a) Component stencil.

(b) Crank-Nicolson stencil.

Figure 6.2: Maximal number of recursive calls for different instances per node for the component stencil (a) and the Crank-Nicolson stencil (b).



(a) General five-point stencil with 3 hops in the first dimension.



(b) General five-point stencil with 3 hops in the last dimension.

Figure 6.3: Maximal number of recursive calls for different instances per node for the general five-point stencil with 3 hops in the first dimension (a) and in the last dimension (b).

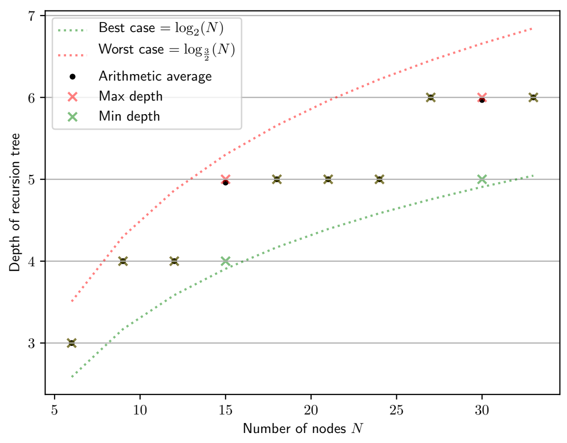


Figure 6.4: Maximal number of recursive calls for different instances per node for the diagonal stencil

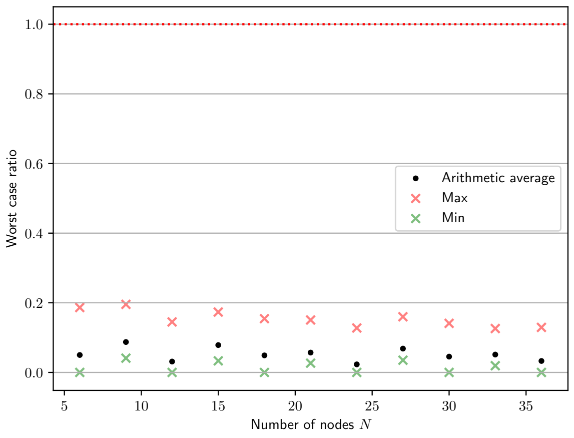
6.2.2 Shifts of the Hyperplane

In order to visualize how many dimension shifts are made over the recursion, we summed up the total amount of shifts done for each process, extracted the maximum number of shifts over all processes $\text{shifts}_{\text{total}}$ and divided it by the theoretical worst case number of possible shifts, that is,

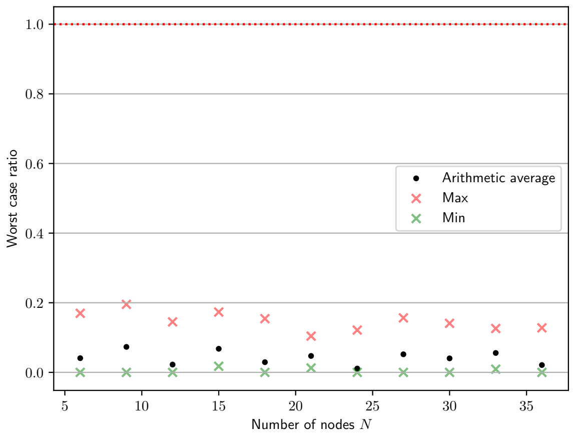
$$\text{Worst case ratio} = \frac{\text{shifts}_{\max}}{\log_{\frac{3}{2}}(N)\sum_{i=1}^{d}\frac{d_i}{2}}. \tag{6.1}$$

Note that by shift we mean actual decrements of the split position as in Line 14 in Algorithm 8, not the initial positioning. The cost of shifting the hyperplane is given by the sum in Equation (5.30) and as it is done over each recursion level contributes significantly to the total runtime. We are interested to see if the practical results are close to the theoretical worst case scenario.

We did this for all the stencils and grid configurations. We plot the maximum, minimum and the average of worst case ratios over the different instances per number of nodes for the general five- and nine-point stencil in Figure 6.5, for the component and Crank-Nicolson in Figure 6.6, for the general five-point stencil with 3 hops in the first and large dimension in Figure 6.7 and in Figure 6.8 for the diagonal stencil. The results indicate that on average the hyperplane algorithm performs less than 10% of the theoretical worst case number of hyperplane shifts.

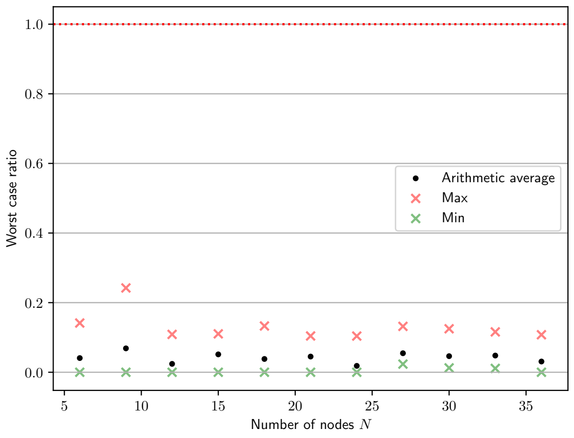


(a) General five-point stencil.

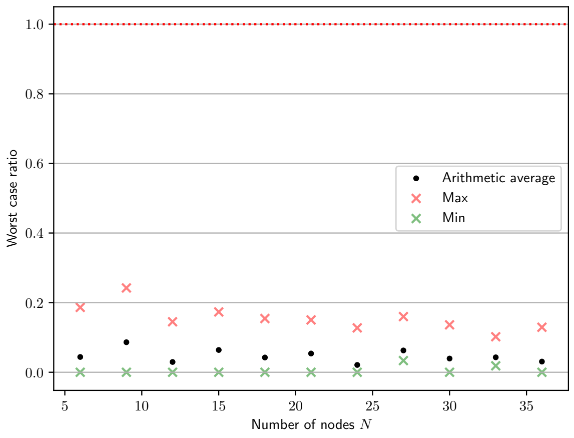


(b) General nine-point stencil.

Figure 6.5: Worst case ratio, Equation (6.1) for different instances per node for the general five-point (a) and nine-point stencil (b).

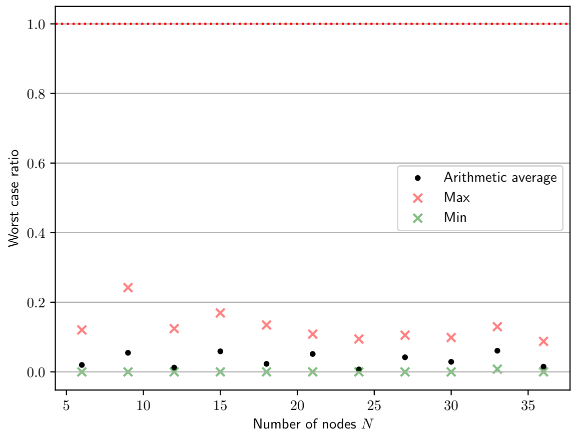


(a) Component stencil.

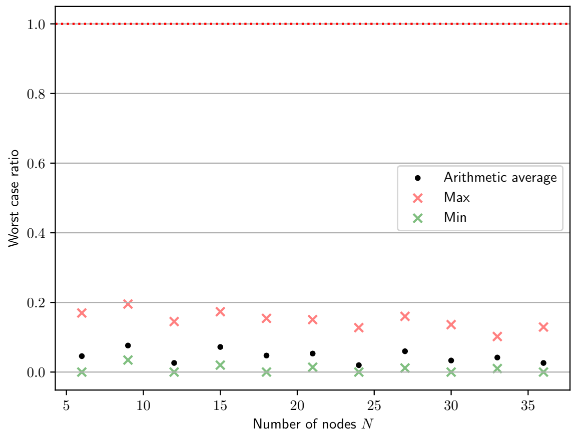


(b) Crank-Nicolson stencil.

Figure 6.6: Worst case ratio, Equation (6.1) for different instances per node for the component (a) and Crank-Nicolson stencil (b).



(a) General five-point stencil with 3 hops in the first dimension.



(b) General five-point stencil with 3 hops in the last dimension.

Figure 6.7: Worst case ratio, Equation (6.1) for different instances per node for the general five-point stencil with 3 hops in the first (a) and the last dimension (b).

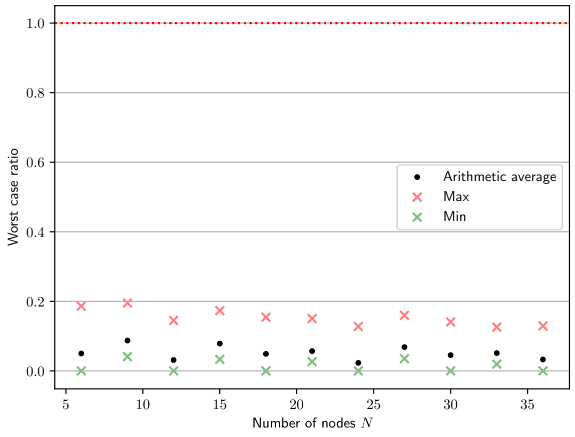


Figure 6.8: Worst case ratio, Equation (6.1) for different instances per node for the diagonal stencil.

6.3 Amount of Inter-Node Communication

In this section, we count the total and maximum amount of inter-node communication edges between the computation nodes for the different reordering schemes, including `MPI_Cart_create` with reordering flag and compare it to the amount of inter-node communication created by `MPI_Cart_create` without reordering, on the different base communicators. The experiments have been done with three different base communicators, i.e., an initial placement of the ranks on the computation nodes. The base communicators where the standard, consecutive rank assignment of `MPI`, a rank placement onto the nodes in a round-robin manner and a random rank assignment.

We will also compare the results obtain by the schemes presented in this thesis to the VieM Mapping Tool [33, 34]. Recall Section 3.2.2 in Chapter 3 for a remainder on how VieM works. In order to be comparable, we model the hardware system by specifying in the input string to be the number of processes per node p_{node} and the number of nodes N, i.e., $s = p_{\text{node}} : N$, while assuming that the distance on a node is 0 and the distance between the nodes is 1, i.e., $c = 0 : 1$. We are only interested in inter- and intra-node communication and assume that the communication between entities in the same level of the hierarchy is equally fast and with the chosen parameters, can directly measure the amount of inter-node communication. VieM outputs the obtained value for the objective function defined in Equation (1.1) in Chapter 1 and therefore we did not have a value for the bottleneck computation node (the node with maximal number of outgoing communication edges) and omitted the comparison.

We define the improvement of a rank reordering scheme over different base mapping in the following way. Let $C_{\text{base, total}}$ and $C_{\text{base, max}}$ be the total amount of inter-node communication cost and the amount of inter-node communication for the node with maximal outgoing communication edges of the base communicator and $C_{X,\text{ total}}$ and $C_{X,\text{ max}}$ the total amount of inter-node communication and the amount of inter-node communication for the node with maximal outgoing communication edges of the rank reordering algorithm X then we define the improvement to be

$$\frac{C_{\text{base, total}}}{C_{X,\text{ total}}} \quad \text{and} \quad \frac{C_{\text{base, max}}}{C_{X,\text{ max}}}. \tag{6.2}$$

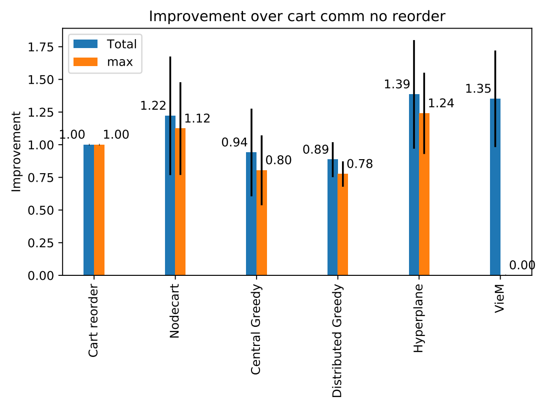
Equation (6.2) shows the increase of the inter-node communication cost in the base rank assignment scheme over the rank reordering techniques.

We measured the amount of inter-node communication by first creating a Cartesian communicator with the different reordering schemes, if any. Given the Cartesian communicator, we created a N communicators consisting of the processes on the same node, i.e., there are N communicators of size p_{node}. Given the Cartesian communicator and the node communicators, we can use a function called `MPI_Group_translate_ranks` in `MPI` that maps the ranks of the processes of one communicator to the other. With this function and the stencil, each process can count the number of communication neighbors on the same node. We aggregated the values over all processes in order to get the sum and the maximum per node.

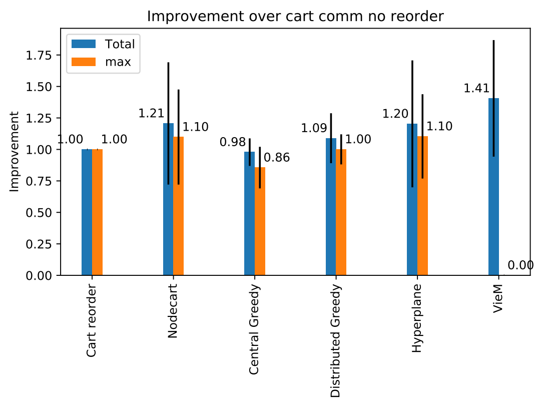
6.3.1 `MPI`'s Default Rank Order

Here, we present the improvement of inter-node communication neighbors over the default mapping of `MPI`. Unfortunately, we cannot compare the algorithms presented in this thesis to VieM for the Crank-Nicolson stencil, since it requires the input graph to be undirected which is not the case. If we were to bypass this inconveniance by making the stencil undirected, we change the structure of the stencil making it hard to compare to the schemes presented in this thesis. For this reason, we have decided to exclude VieM for the Crank-Nicolson stencil. In Figures 6.9, 6.10, 6.11 and 6.12, we averaged[2] the results over all instances and plot the improvement defined in Equation (6.2) for the total (blue) and the maximal (orange) number of inter-node communication. The black line depicts the standard deviation. Overall we can see that the hyperplane approach finds on average better or equal mappings to Gropp's [13, 14] approach. It is even capable to find better partitions than VieM for the general five-point stencil. This is quite remarkable, since the runtime of the hyperplane algorithm does not explicitly depend on the number of nodes and vertices in the graph like VieM and thus finds better partitions in shorter time. Unfortunately, both greedy approaches perform badly for all but the component and the diagonal stencil. We can also see that *Cart reorder* does not yield any kind of improvements over the non reordering version of `MPI_Cart_create`. For the component stencil, `Nodecart` was able to find partitions with *zero* inter-node communication cost for 4 out of 331 instances. The distributed greedy algorithm found 16 instances with zero inter-node communication, whereas the centralized, the hyperplane and VieM found 19 partitions with zero inter-node communication cost. This was the case, for all tested base communicators.

[2]We used the geometric average.

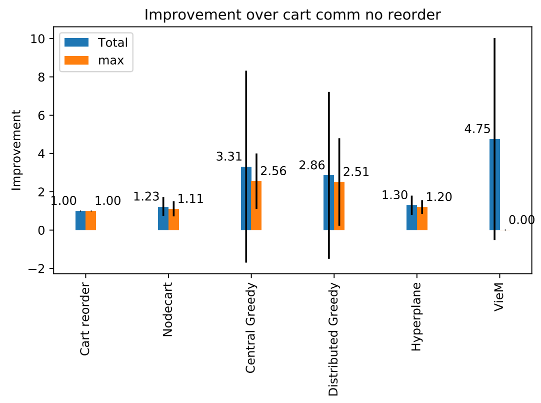


(a) General five-point stencil

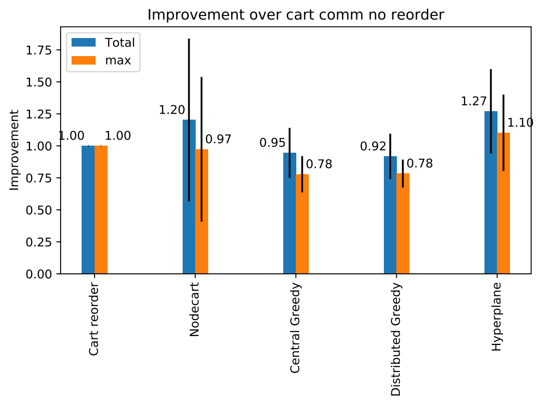


(b) General nine-point stencil

Figure 6.9: Improvement defined in Equation (6.2) over `MPI`'s default mapping for the general five- (a) and nine-point stencil (b).

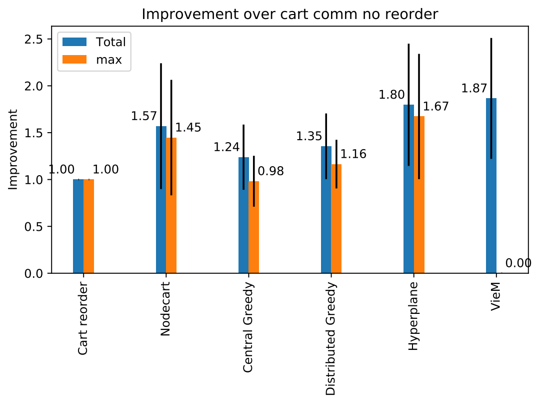


(a) Diagonal stencil

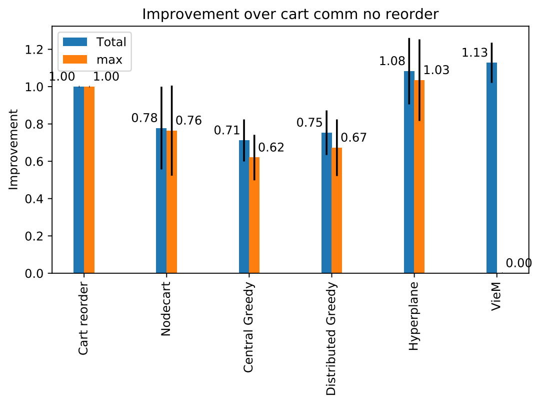


(b) Crank-Nicolson stencil

Figure 6.10: Improvement defined in Equation (6.2) over `MPI`'s default mapping for the diagonal (a) and Crank-Nicolson stencil (b).



(a) General five-point stencil with 3 hops in the first dimension



(b) General five-point stencil with 3 hops in the last dimension

Figure 6.11: Improvement defined in Equation (6.2) over `MPI`'s default mapping for the general five-point stencil with 3 hops in the first (a) and 3 hops in the last dimension (b).

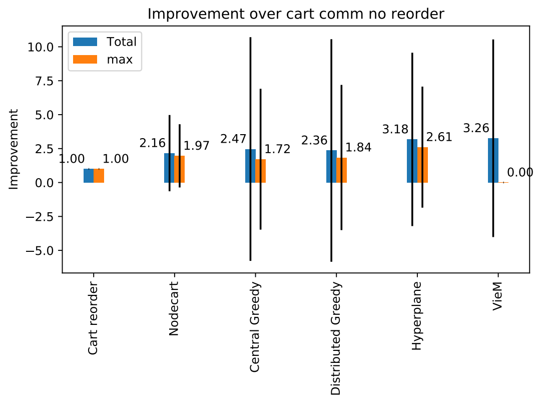
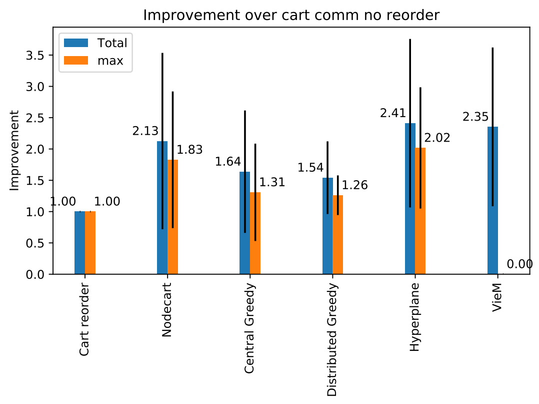


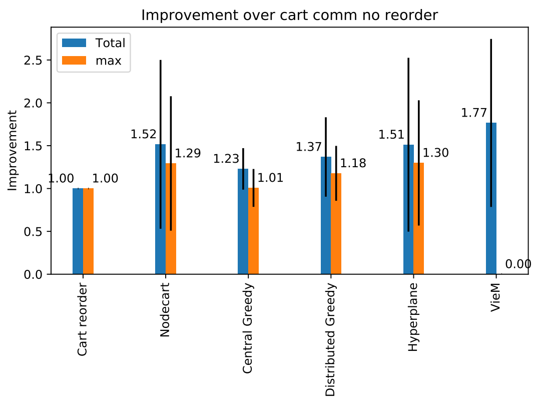
Figure 6.12: Improvement defined in Equation (6.2) over `MPI`'s default mapping for the component stencil.

6.3.2 Round-Robin Rank Ordering

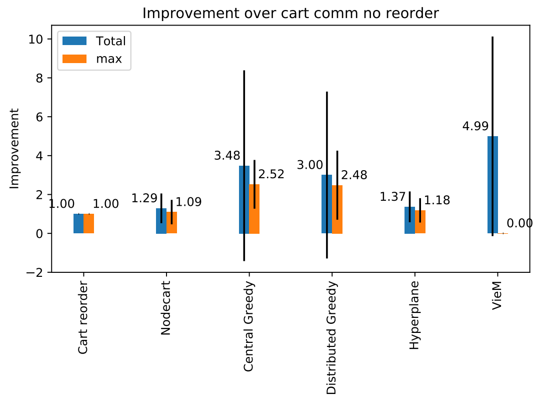
In Figures 6.13, 6.14 and 6.15 we plot the improvement, for all algorithms over the Cartesian communicator instantiated with `MPI_Cart_create` on a base communicator with a round-robin assignment scheme. Since the initial mapping is far away from the optimal, all schemes improve the amount of inter-node communication. We can see that the relative performance between the different approaches is similar as in the default initial rank order scheme.



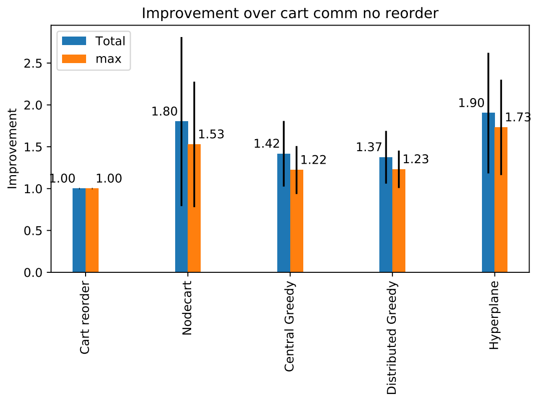
(a) General five-point stencil



(b) General nine-point stencil

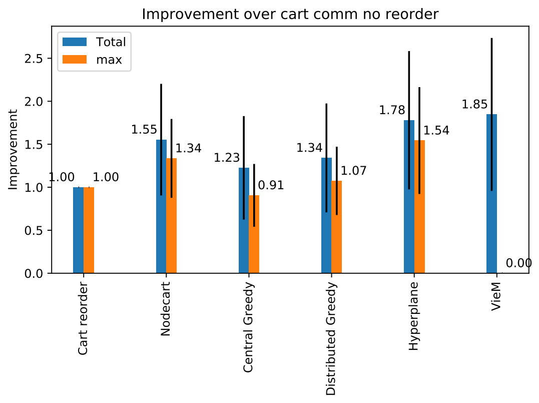


(c) Diagonal stencil

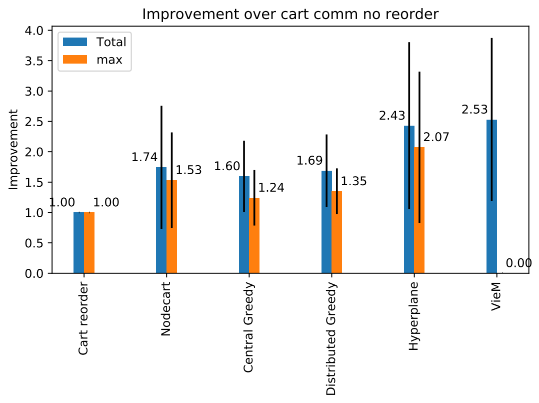


(d) Crank-Nicolson stencil

Figure 6.13: Improvement defined in Equation (6.2) over an initial round-robin mapping for the general five- (a), nine-point (b), the diagonal (c) and Crank-Nicolson stencil (d).



(a) General five-point stencil with 3 hops in the first dimension



(b) General five-point stencil with 3 hops in the last dimension

Figure 6.14: Improvement defined in Equation (6.2) over an initial round-robin mapping for the general five-point stencil with 3 hops in the first (a) and 3 hops in the last dimension (b).

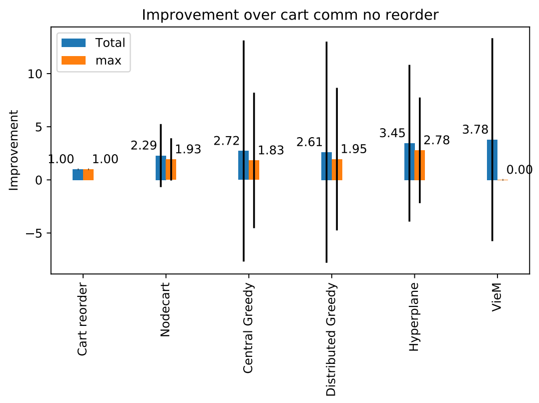
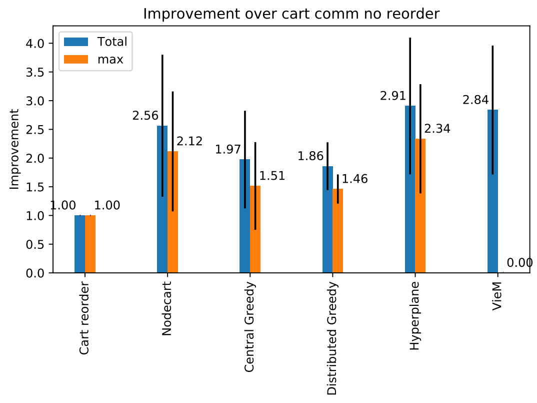


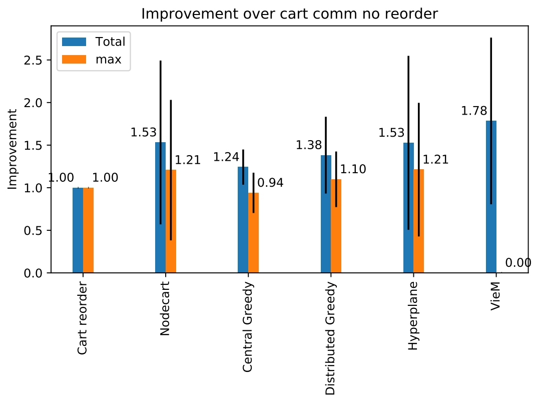
Figure 6.15: Improvement defined in Equation (6.2) over an initial round-robin mapping for the component stencil.

6.3.3 Random Rank Ordering

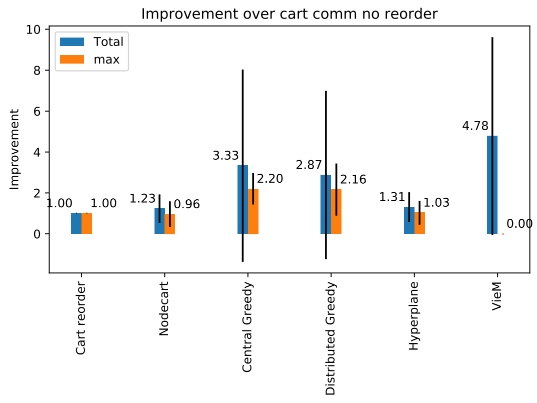
Finally, we repeated the experiment with a random base communicator. The improvements can be seen in Figures 6.16, 6.17 and 6.18. The improvement is even better than for the round-robin base communicator, since the round-robin scheme is not completely oblivious to the hardware, but the ratios between the different schemes are approximately the same.



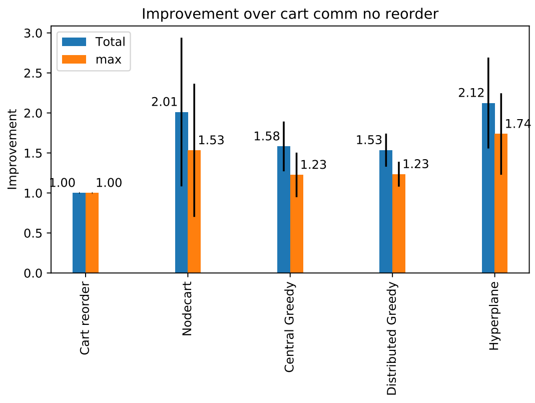
(a) General five-point stencil



(b) General nine-point stencil

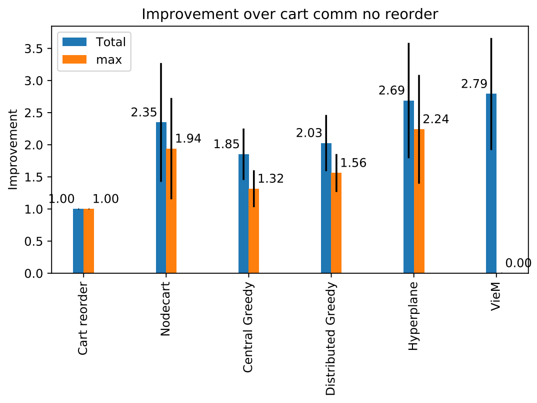


(c) Diagonal stencil

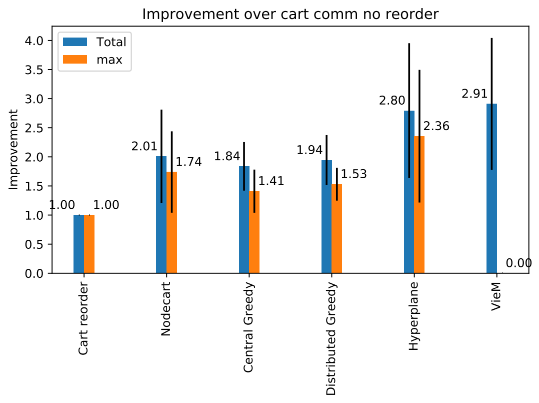


(d) Crank-Nicolson stencil

Figure 6.16: Improvement defined in Equation (6.2) over an initial random mapping for the general five- (a), nine-point (b), the diagonal (c) and Crank-Nicolson stencil (d)



(a) General five-point stencil with 3 hops in the first dimension



(b) General five-point stencil with 3 hops in the last dimension

Figure 6.17: Improvement defined in Equation (6.2) over an initial random mapping for the general five-point stencil with 3 hops in the first (a) and 3 hops in the last dimension (b).

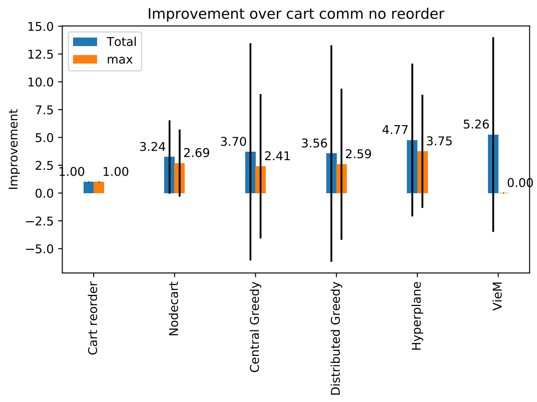


Figure 6.18: Improvement defined in Equation (6.2) over an initial random mapping for the component stencil.

6.4 Improvements in Communication Time

In order to verify that the improvement over the default and Gropp's mapping [13, 14] also benefits the communication in terms of time, we benchmarked the communication time needed for a stencil all-to-all communication, for different reordering schemes. The experiment consisted of a stencil sweep, in which all processes sent and received data of various size to and from their neighbors given by the stencil. The exchange was done with the `MPI_Neighbor_alltoall` routine explained in Section 1.2.4 in Chapter 1. The processes were synchronized with the `MPI_Barrier` function before measuring the time of the message exchange. Processes exchanged data over 1500 iterations and in each iteration we captured the minimal from all maximum times and the average time over all processes. We do this, since we are not only interested in minimizing the maximum offnode communication over all the nodes, but we want to reduce the total amount of inter-node communication. The maximum communication time needed is strongly biased towards the bottleneck, since it is the node with the largest amount of offnode communication. The QAP problem formulation does not explicitly involve reducing the maximum number of offnode communication partners. In order to prevent cold start issues, we initialized 3 initial warm-up runs, which do not contribute to the experiment. We calculated the minimal of the maximum and the average time in the following way: Let t_{max} and t_{avg} denote the maximum and the average time over all processes for a run, i.e.,

$$t_{\text{avg}} = \sum_{i=1}^{P} \frac{t_i}{P}, \tag{6.3}$$

where P is the number of processes then the minimal maximum $\max_{\text{min}}$ and the average of average μ_{avg} for an experiment is defined as

$$\begin{aligned} \max_{\text{min}} &= \min(t_{\text{max}},\ \max_{\text{min}}) \\ \mu_{\text{avg}} &= \frac{\sum_{\text{exchanges}} t_{\text{avg}}}{\text{exchanges}} \end{aligned} \tag{6.4}$$

where *exchanges* is the number of exchanges. The code snippet for this experiment can be seen in Listing 6.2.[3]

We plot the bandwidth for different stencils and different rank reordering schemes for grid instances with 33 nodes, 32 processes per node and 3 dimensions. The amount of total and maximum inter-node communication can be found in Tables

[3] `MPI_Wtime` returns the elapsed seconds after a time-point in the past.

Table 6.1: Total number of inter-node communication edges for different reordering schemes

Rank Reordering Scheme	Stencil types						
	five point	nine point	component	five point hops last	diagonal	five point hops first	crank nicolson
cart	2416	16324	2416	2416	6160	5760	4530
cart reorder	2416	16324	2416	2416	6160	5760	4530
nodecart	2272	15928	2272	2272	6160	4032	4260
central greedy	2402	16166	688	3998	2872	4228	4502
dist greedy	2660	13572	888	3626	3012	3946	4863
hyperplane	1552	12544	944	1888	4000	2592	2880

Table 6.2: Max number of inter-node communication edges for different reordering schemes

Rank Reordering Scheme	Stencil types						
	five point	nine point	component	five point hops last	diagonal	five point hops first	crank nicolson
cart	80	572	80	80	224	208	150
cart reorder	80	572	80	80	224	208	150
nodecart	80	572	80	80	224	160	150
central greedy	127	638	48	162	140	220	197
dist greedy	118	558	55	155	136	159	193
hyperplane	80	539	72	80	224	112	150

6.1 and 6.2. One can see the average bandwidth in Bytes/s of the different schemes for the mentioned constellation in Figure 6.19 for the five- and nine-point stencil, in Figure 6.20 for the diagonal and the Crank-Nicolson stencil, in Figure 6.21 for the general five-point stencil with 3 hops in the first and last direction and in Figure 6.22 for the component stencil. The performance is in general proportional to the total number of inter-node communication. The hyperplane algorithm finds partitions with the lowest amount inter-node communication for the five-point, the nine-point, the five-points with hops in the first and last direction and the Crank-Nicolson stencil. Astoundingly, the distributed greedy algorithm has about 8% more total inter-node communication than the hyperplane algorithm, but has an increased in average bandwidth of up to 19% over the bandwidth induced by the hyperplane algorithm. Both greedy approaches did not perform well for the general five-point stencil, with or without hops, but could further improve the bandwidth for the diagonal stencils which was designed to be difficult to partition well for the hyperplane algorithm. Interestingly, the bandwidth seemed to be improving inconsistently for only some reordering schemes and messages sizes larger than $\approx$ 14000 bytes.

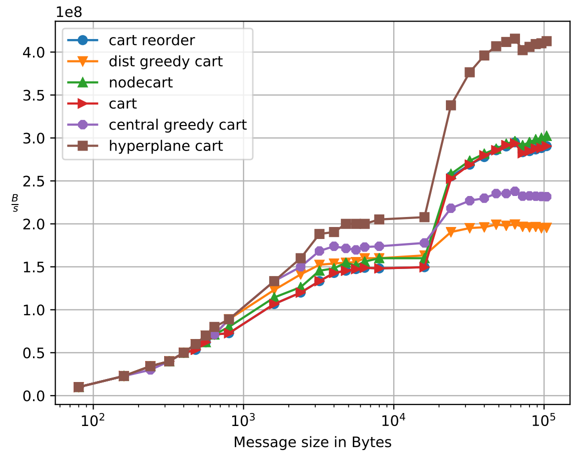
The minimal maximum $\max_{\text{min}}$ bandwidth can be seen in Figures 6.23, 6.24 6.25 and 6.26. By comparing the amount of inter-node communication for the bottleneck node, we can see that the hyperplane approach yielded mappings with less or equal amount of maximal off-node communication than `Nodecart` or the two mappings of `MPI`, for all stencils. The two greedy approaches could outperform the hyperplane algorithm in the case of the component and the diagonal stencil.

For the diagonal stencil, the hyperplane algorithm has 65% more maximal inter-node communication edges than the distributed greedy appraoch, which resulted in an increase of up to a factor of 3 in bandwidth.

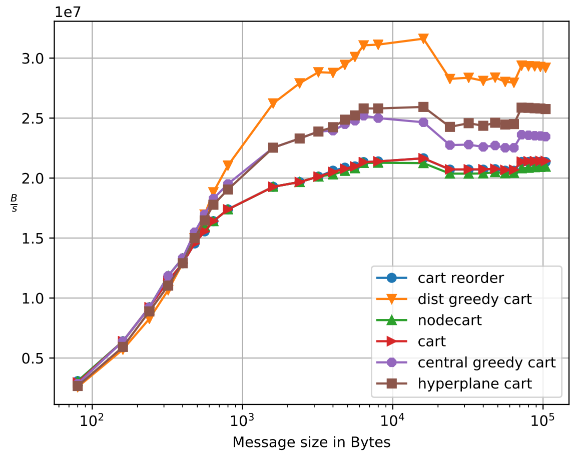
The standard deviation for the mean neighbor all-to-all exchange can be seen in Appendix A, in Tables A.8, A.9, A.10, A.11, A.12, A.13, A.14.

Listing 6.2: Code snippet for the communication exchange benchmark

```
//Loop over the exchanges
for( int ex = 0; ex < n_exchanges; ex++ ){
        //Synchronize and measure the time
        MPI_Barrier(dist_graph_comm);
        start = MPI_Wtime();
        MPI_Neighbor_alltoall(send_buff, size,
                MPI_DOUBLE, recv_buff, size,
                MPI_DOUBLE, dist_graph_comm);
        end = MPI_Wtime();

        //check if message is correct
        checkup( ... )

        ellapsed = end - start;

        //Reduce the obtained result
        MPI_Allreduce(&ellapsed, &one_run_max, 1,
                MPI_DOUBLE, MPI_MAX,
                dist_graph_comm);
        MPI_Allreduce(&ellapsed, &one_run_avg, 1,
                MPI_DOUBLE, MPI_SUM,
                dist_graph_comm);

        //save the values and average the time over
        //all provesses for run_avg
        min_max_time = std::max(one_run_max, min_max_time);
        run_avg +=
                one_run_avg/(w_size*n_exchanges);
}
```

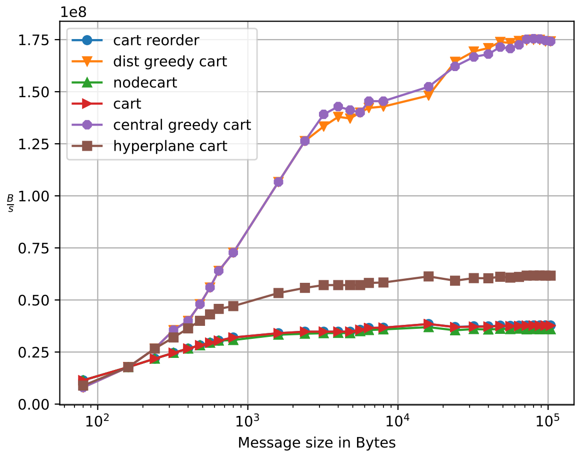


(a) General five-point stencil

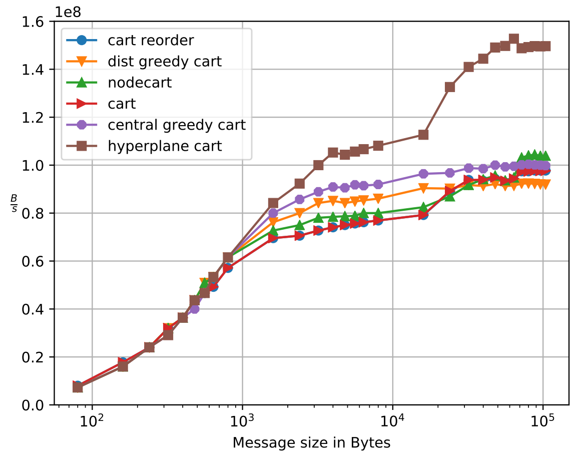


(b) General nine-point stencil

Figure 6.19: Bandwidth for `MPI_Neighbor_alltoall` of different algorithms for the general five- (a) and nine-point stencil (b).

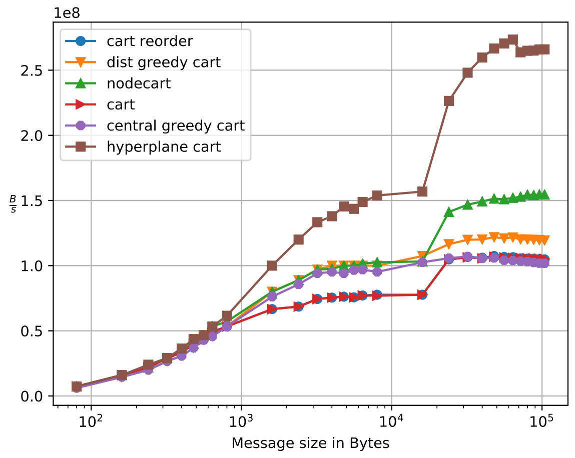


(a) Diagonal stencil

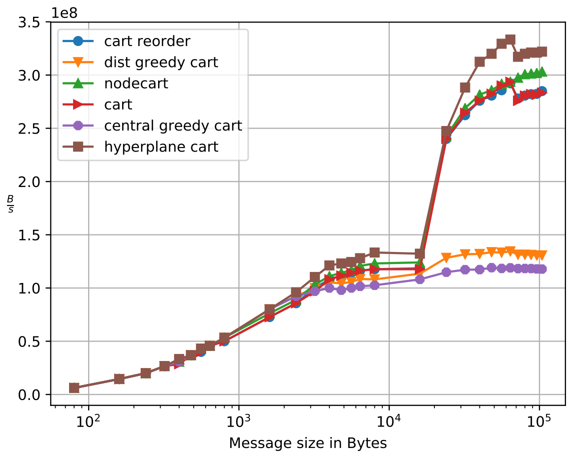


(b) Crank-Nicolson stencil

Figure 6.20: Bandwidth for `MPI_Neighbor_alltoall` of different algorithms for the diagonal (a) and Crank-Nicolson stencil (b).



(a) General five-point stencil with 3 hops in the last dimension



(b) General five-point stencil with 3 hops in the first dimension

Figure 6.21: Bandwidth for `MPI_Neighbor_alltoall` of different algorithms for the general five-point stencil with 3 hops in the first (a) and 3 hops in the last dimension (b).

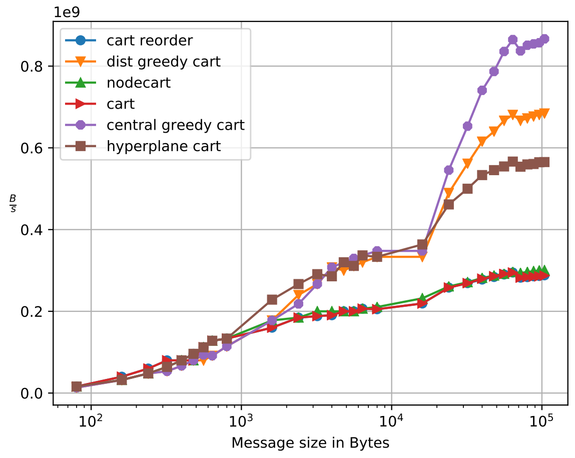
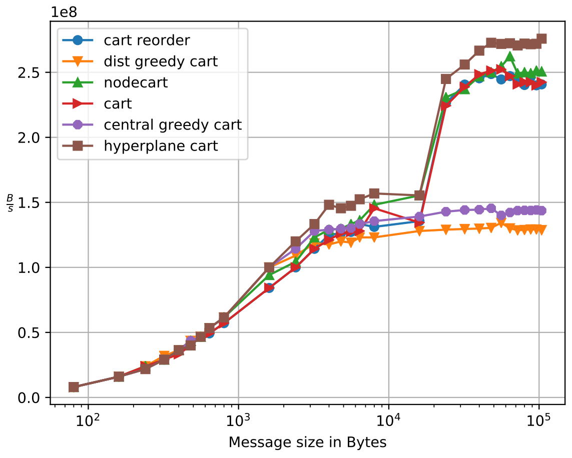
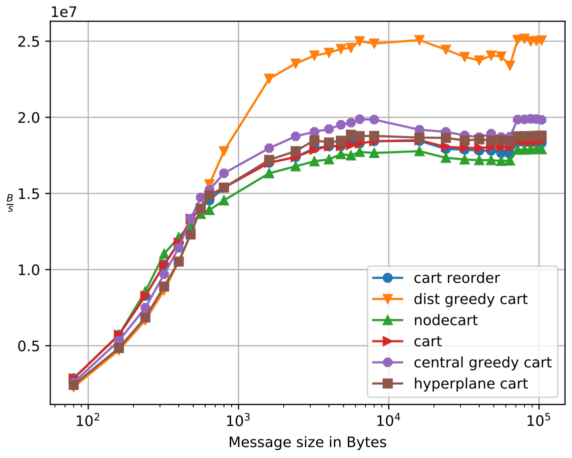


Figure 6.22: Bandwidth for `MPI_Neighbor_alltoall` of different algorithms for the component stencil.

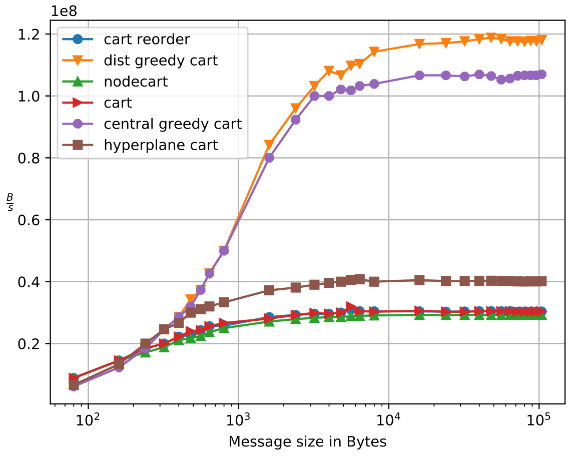


(a) General five-point stencil

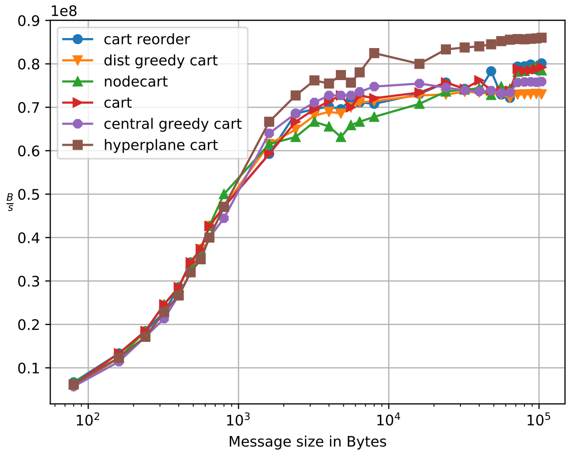


(b) General nine-point stencil

Figure 6.23: Minimal maximum bandwidth for `MPI_Neighbor_alltoall` of different algorithms for the general five- (a) and nine-point stencil (b).

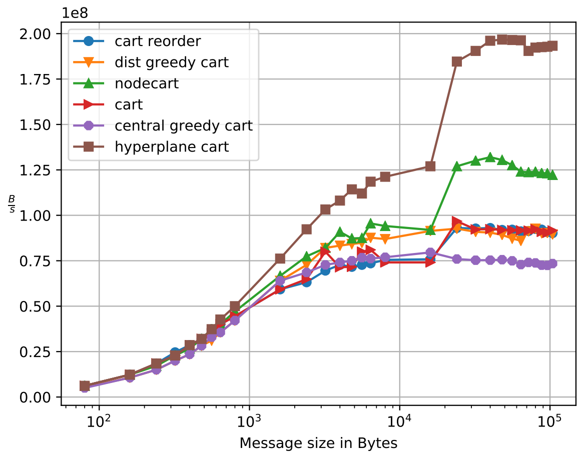


(a) Diagonal stencil

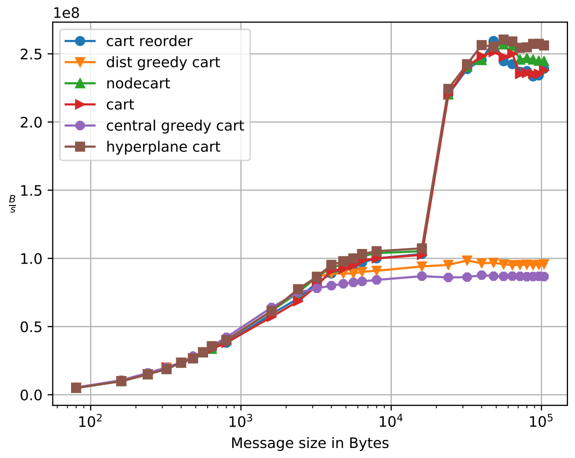


(b) Crank-Nicolson stencil

Figure 6.24: Minimal maximum bandwidth for `MPI_Neighbor_alltoall` of different algorithms for the diagonal (a) and Crank-Nicolson stencil (b).



(a) General five-point stencil with 3 hops in the last dimension



(b) General five-point stencil with 3 hops in the first dimension

Figure 6.25: Minimal maximum bandwidth for `MPI_Neighbor_alltoall` of different algorithms for the general five-point stencil with 3 hops in the first (a) and 3 hops in the last dimension (b).

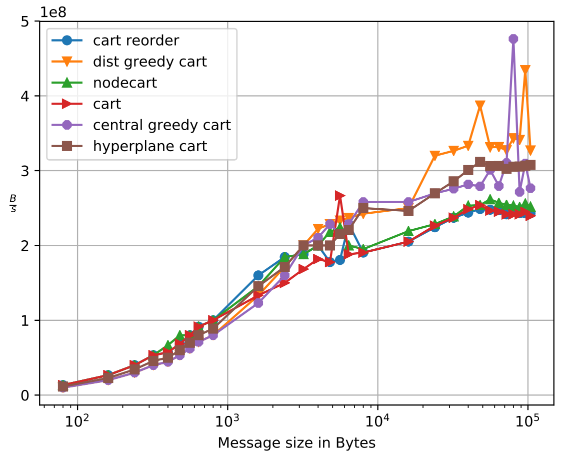
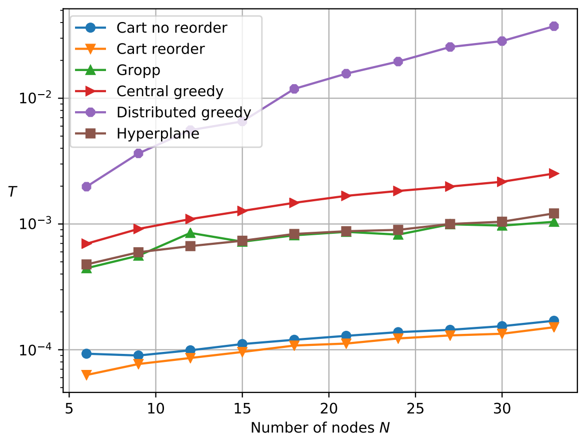


Figure 6.26: Minimal maximum bandwidth for `MPI_Neighbor_alltoall` of different algorithms for the component stencil.

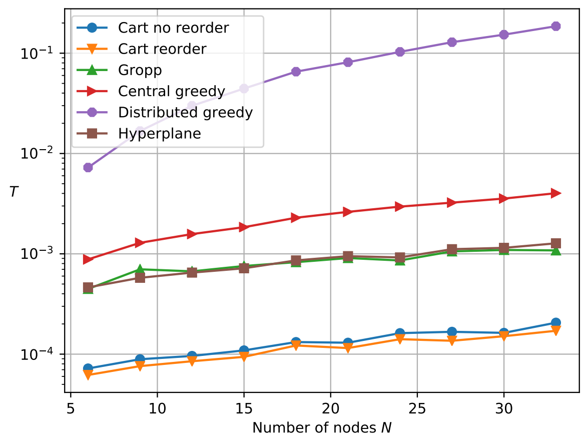
6.5 Instantiation Time

In this section, we measure the instantiation time T in seconds for the different, presented algorithms and stencils. We repeated the experiments 50 times and always used the maximum time a process needed for the instantiation. We show the instantiation time T with varying number of nodes and a fixed number of processes per node (32) and number of dimensions (3) in Figure 6.27 for the general five- and nine-point stencil, in Figure 6.28 for the diagonal and the Crank-Nicolson stencil, in Figure 6.29 for the five-point stencils with 3 hops in the first and last dimension and in Figure 6.30 for the component stencil. Note the exponential scale of the time axis. The standard deviation can be found in the Tables A.1, A.2, A.3, A.4, A.5, A.6, A.7. The time was measured for the complete instantiation of the Cartesian communicators. To be more precise, we instantiated a new Cartesian communicator with the `MPI_Cart_create` routine with the new, calculated rank order if any. We measured the whole process of instantiation, i.e., the calculation of the reordering and the instantiation of the communicators, thus in order to compare only the runtime for the reordering schemes, we can subtract the time needed for a Cartesian communicator instantiated in the default way of `MPI` without reordering. One can see that the two versions of `MPI_Cart_create` are almost by an order of magnitude faster than the other schemes. This indicates once more, that the reorder flag in the `MPI_Cart_create` does not perform any reordering. For the small instances, the runtime of the hyperplane algorithm matches that of `Nodecart`, but produces better mappings. The distributed greedy approach is by far the most expensive, due to its initial BFS and the final rank assignment control routine.[4]

[4]During the experiments, we encountered problems with a node and had to exclude the instance with 36 computation nodes.

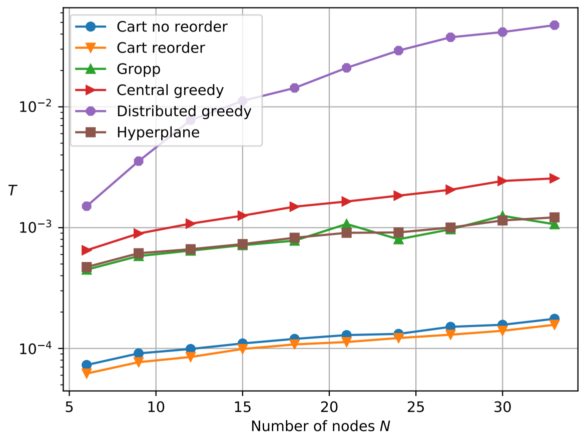


(a) General five-point stencil

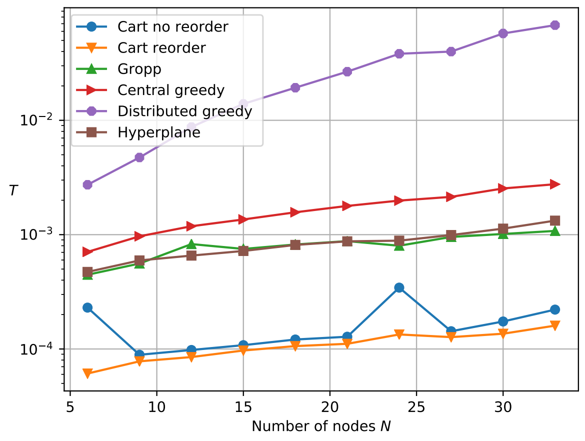


(b) General nine-point stencil

Figure 6.27: Instantiation time of different algorithms for the general five- (a) and nine-point stencil (b).

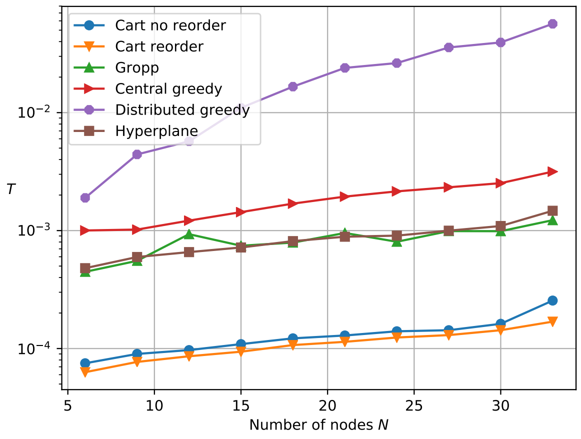


(a) Diagonal stencil

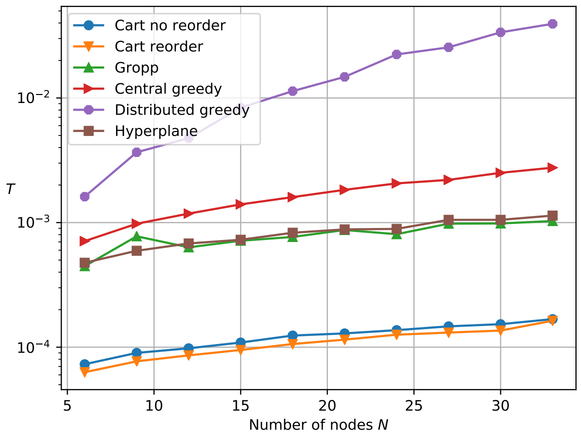


(b) Crank-Nicolson stencil

Figure 6.28: Instantiation time of different algorithms for the diagonal (a) and Crank-Nicolson stencil (b).



(a) General five-point stencil with 3 hops in the last dimension



(b) General five-point stencil with 3 hops in the first dimension

Figure 6.29: Instantiation time of different algorithms for the general five-point stencil with 3 hops in the first (a) and 3 hops in the last dimension (b).

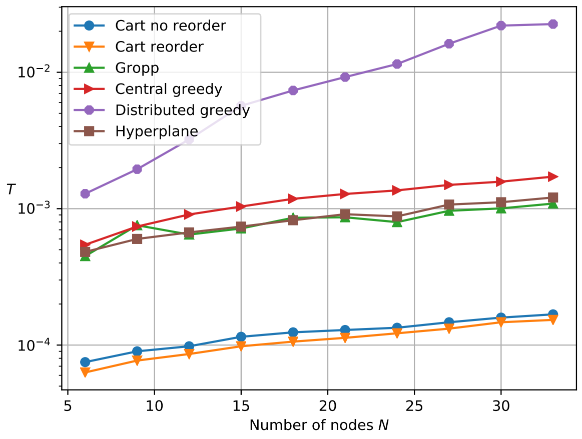


Figure 6.30: Instantiation time of different algorithms for the component stencil.

Chapter 7

Conclusions

We conclude the thesis in this section, with a brief summary of what was done over the chapters. Finally, we give an outlook on some possible future work.

7.1 Summary

The problem of inter-node communication reduction is important, since it can significantly improve the performance of parallel applications. Several attempts were made to efficiently map Cartesian topologies to parallel machines with hierarchical communication performance. In this thesis we attempted to derive approaches in `MPI` for mapping Cartesian graphs induced by isomorphic communication on Cartesian grids onto computation nodes s.t. the total number of inter-node communication is reduced.

In Chapter 1 we provided the necessary vocabulary and definitions for the reader and introduced `MPI` and some of its routines. We defined exactly what we mean with isomorphic communication and Cartesian graphs and introduced some necessary functions which are valuable for understanding the algorithms and the experiments conducted in Chapter 6.

Chapter 2 provided the motivation for this thesis. We showed examples for the increase of intra-node communication. Further, we introduced a problem formulation and defined what the objective was, namely to find an algorithm that experimentally outperforms Gropp's approach, while having a fast theoretical runtime.

We continued in Chapter 3 by giving an overview of relevant research of the mapping problem. We reported some approaches developed specifically for `MPI` and other general graph mapping techniques. More importantly, we introduced Gropp's `Nodecart` [13, 14] approach and the Vienna Mapping tool developed by

Schulz and Träff [33, 34] since we compared our approaches in terms of inter-node communication reduction to them.

The two greedy approaches were presented in Chapter 4. We showed the concept of the greedy rank assignment of the processes to the computation nodes using a priority queue. This approach was quite simple, but centralized and thus did not scale well. For that purpose we introduced the distributed greedy approach which performed initial BFS runs in order to find ranks that have maximal distance to one another in the graph theoretical sense. Each of those ranks were assigned to a computation node and were responsible to fill it up, using the same greedy approach as the centralized one. The theoretical runtime of these two approaches was still in the number of vertices and edges of the input graph, which was too slow for our objective.

In Chapter 5 we introduced the hyperplane algorithm, which recursively splits the input Cartesian graph along a grid dimension, while trying to maximize parallelism between the cutting hyperplane and the overall stencil communication. The big advantage of this approach was that each process could calculate its new rank without the need for communication. We could show, that it was always possible to find these cuts and could derive a theoretical runtime that is no longer dependent on the number of vertices and edges of the input graph.

The experimental evaluation of the presented schemes was done Chapter 6. There, we first benchmarked the behavior of the runtime of the hyperplane approach in terms of recursion tree depth and number of hyperplane shifts. Further, we compared the amount of improvement the different schemes provide over the amount of inter-node communication induces by `MPI_Cart_create` for different initial rank assignments. We tested this for a variety of stencils and grid configurations and could show, that for the tested instances the hyperplane algorithm was on average able to outperform Gropp's `Nodecart` approach. In case of the general five-point stencil, it was even able to outperform the general graph mapping software VieM. The two greedy approaches were only beneficial for some stencils and for others were even worse in terms of inter-node communication than `MPI_Cart_create`. We further investigated the implications of the reduction of inter-node communication for a neighbor all-to-all routine. We could show for a specific grid instance, that the hyperplane algorithm indeed resulted in an increase of bandwidth, even though we saw inconsistent patterns for the distributed greedy approach and sudden bursts of bandwidth for certain message sizes for some reordering schemes. The chapter is concluded with a comparison of instantiation time for a specific grid instance and different stencils. We could show for the small input graphs, that the hyperplane approach was approximately as fast as Gropp's `Nodecart` algorithm. Thus, by showing that we could further reduce the amoung inter-node communication than Gropp for arbitrary isomorphic communication patterns and compete with him in terms of runtime, we achieved the initially stated goal.

7.2 Future Work

Firstly, we can extend the hyperplane algorithm to be able to be applied to systems with different node sizes. This can be done by initially performing more work in exploring and communicating the different nodes sizes and keeping track of those sizes while assigning the ranks for the greedy approaches. The hyperplane algorithm could partition the Cartesian graph using the smallest or largest node size and then map the partitions consecutively to the computation nodes. This approach is not ideal, since some partitions can be spread across multiple nodes, but allows using asymmetrical systems and can be helpful. It would be interesting to test the approaches on bigger machines incorporating more nodes and processes per node.

As further possible improvement in terms of inter-node communication cost for the hyperplane algorithm, we could position the hyperplane in a boundary aware manner. That is, instead of moving the hyperplane in the decreasing dimension direction, we can push it towards the borders of the original grid. Intuitively, if the grid is non-periodic, this could improve the total and maximal number of inter-node communication edges. Since the direction of hyperplane shift is only dependent on the position of the subgrid in the original grid, this position awareness would not diminish the runtime.

Additionally, we can extend our approaches to handle weighted Cartesian graphs C. The weights of the communication edges could represent message sizes or communication frequency. By incorporating this information in the stencil, we can adapt the objective function that determines the split quality of a dimension for the hyperplane algorithm or the priority of a neighbor in the greedy approaches.

At last, we could extend all the methods to be applicable to multiple hierarchy levels and reorder the ranks accordingly, trying to minimize the inter-entity communication on each hardware level. This could be done with hierarchy information extraction tools, such as HWLOC or by the different types in Open Mpi 4.0.1 for the `MPI_Comm_split_type` routine, should they be implemented.

Appendix A

Appendix

A.1 Standard Deviation for Instantiation Time

Table A.1: Standard deviation of instatiation time in seconds for the five point stencil, 32 processes per node and 3 dimensions

Reordering Scheme	**Number of nodes**									
	6	9	12	15	18	21	24	27	30	33
Cart no reorder	$1.71e-04$	$5.90e-05$	$5.40e-05$	$6.60e-05$	$6.30e-05$	$8.20e-05$	$7.10e-05$	$8.40e-05$	$8.30e-05$	$8.20e-05$
Cart reorder	$6.00e-06$	$5.00e-06$	$6.00e-06$	$1.00e-05$	$8.00e-06$	$7.00e-06$	$1.40e-05$	$1.30e-05$	$8.00e-06$	$1.60e-05$
Gropp	$5.46e-04$	$7.57e-04$	$2.47e-03$	$1.22e-03$	$1.33e-03$	$1.39e-03$	$9.69e-04$	$1.64e-03$	$1.33e-03$	$1.23e-03$
Central greedy	$4.00e-04$	$5.05e-04$	$4.91e-04$	$5.03e-04$	$5.63e-04$	$7.29e-04$	$6.77e-04$	$5.40e-04$	$5.50e-04$	$7.19e-04$
Distributed greedy	$5.82e-04$	$6.81e-04$	$5.00e-04$	$4.69e-04$	$6.88e-04$	$7.31e-04$	$8.01e-04$	$8.92e-04$	$1.06e-03$	$1.41e-03$
Hyperplane	$2.62e-04$	$3.04e-04$	$4.00e-04$	$4.13e-04$	$5.10e-04$	$5.08e-04$	$3.95e-04$	$4.63e-04$	$4.57e-04$	$4.59e-04$

Table A.2: Standard deviation of instatiation time in seconds for the nine point stencil, 32 processes per node and 3 dimensions

Reordering Scheme	**Number of nodes**									
	6	9	12	15	18	21	24	27	30	33
Cart no reorder	$4.90e-05$	$5.30e-05$	$5.60e-05$	$7.00e-05$	$1.10e-04$	$8.60e-05$	$1.18e-04$	$1.20e-04$	$9.90e-05$	$1.78e-04$
Cart reorder	$4.00e-06$	$5.00e-06$	$7.00e-06$	$1.50e-05$	$9.60e-05$	$2.50e-05$	$1.33e-04$	$5.20e-05$	$6.20e-05$	$9.70e-05$
Gropp	$5.60e-04$	$1.77e-03$	$1.23e-03$	$1.35e-03$	$1.16e-03$	$1.45e-03$	$8.84e-04$	$1.64e-03$	$1.34e-03$	$1.13e-03$
Central greedy	$2.72e-04$	$5.76e-04$	$4.64e-04$	$4.08e-04$	$6.43e-04$	$6.41e-04$	$6.87e-04$	$7.16e-04$	$7.57e-04$	$9.04e-04$
Distributed greedy	$3.52e-04$	$7.59e-04$	$6.66e-04$	$7.51e-04$	$8.77e-04$	$1.44e-03$	$1.32e-03$	$1.45e-03$	$1.33e-03$	$1.59e-03$
Hyperplane	$2.45e-04$	$3.08e-04$	$3.57e-04$	$4.28e-04$	$5.34e-04$	$5.69e-04$	$2.62e-04$	$8.05e-04$	$5.51e-04$	$5.33e-04$

Table A.3: Standard deviation of instatiation time in seconds for the diagonal stencil, 32 processes per node and 3 dimensions

Reordering Scheme	Number of nodes									
	6	9	12	15	18	21	24	27	30	33
Cart no reorder	4.60e − 05	5.20e − 05	5.50e − 05	6.90e − 05	6.70e − 05	7.80e − 05	6.50e − 05	8.40e − 05	8.50e − 05	1.02e − 04
Cart reorder	7.00e − 06	4.00e − 06	6.00e − 06	1.90e − 05	1.60e − 05	7.00e − 06	1.20e − 05	1.20e − 05	8.00e − 06	6.40e − 05
Gropp	5.56e − 04	9.23e − 04	1.03e − 03	1.19e − 03	1.17e − 03	2.85e − 03	8.51e − 04	1.45e − 03	1.71e − 03	1.30e − 03
Central greedy	2.65e − 04	3.24e − 04	3.88e − 04	4.06e − 04	6.91e − 04	4.90e − 04	5.11e − 04	7.54e − 04	9.64e − 04	7.90e − 04
Distributed greedy	3.36e − 04	4.09e − 04	5.70e − 04	9.49e − 04	9.96e − 04	1.25e − 03	1.47e − 03	1.81e − 03	2.27e − 03	2.25e − 03
Hyperplane	2.25e − 04	4.10e − 04	4.07e − 04	4.24e − 04	5.23e − 04	6.48e − 04	3.06e − 04	4.82e − 04	7.48e − 04	5.26e − 04

Table A.4: Standard deviation of instatiation time in seconds for the crank nicolson stencil, 32 processes per node and 3 dimensions

Reordering Scheme	Number of nodes									
	6	9	12	15	18	21	24	27	30	33
Cart no reorder	1.17e − 03	5.30e − 05	5.80e − 05	6.20e − 05	6.70e − 05	8.10e − 05	1.55e − 03	7.50e − 05	1.44e − 04	2.64e − 04
Cart reorder	4.00e − 06	1.40e − 05	5.00e − 06	8.00e − 06	8.00e − 06	5.00e − 06	7.50e − 05	9.00e − 06	1.40e − 05	6.70e − 05
Gropp	5.52e − 04	7.42e − 04	2.31e − 03	1.29e − 03	1.45e − 03	1.58e − 03	7.79e − 04	1.39e − 03	1.37e − 03	1.15e − 03
Central greedy	3.27e − 04	4.26e − 04	5.05e − 04	5.30e − 04	6.15e − 04	7.16e − 04	8.12e − 04	6.08e − 04	8.71e − 04	8.62e − 04
Distributed greedy	3.41e − 04	4.50e − 04	5.60e − 04	7.32e − 04	8.68e − 04	8.43e − 04	1.28e − 03	2.18e − 03	1.97e − 03	2.04e − 03
Hyperplane	2.18e − 04	3.10e − 04	3.95e − 04	4.13e − 04	4.97e − 04	5.15e − 04	3.01e − 04	4.54e − 04	6.09e − 04	7.19e − 04

Table A.5: Standard deviation of instatiation time in seconds for the five point hops first stencil, 32 processes per node and 3 dimensions

Reordering Scheme	Number of nodes									
	6	9	12	15	18	21	24	27	30	33
Cart no reorder	4.70e − 05	5.70e − 05	5.70e − 05	7.00e − 05	6.40e − 05	7.60e − 05	6.70e − 05	7.70e − 05	1.03e − 04	3.93e − 04
Cart reorder	5.00e − 06	8.00e − 06	7.00e − 06	7.00e − 06	1.30e − 05	7.00e − 06	1.20e − 05	2.30e − 05	3.70e − 05	1.34e − 04
Gropp	5.53e − 04	7.38e − 04	3.06e − 03	1.37e − 03	1.26e − 03	1.98e − 03	8.27e − 04	1.69e − 03	1.29e − 03	1.31e − 03
Central greedy	2.04e − 03	3.21e − 04	3.83e − 04	4.01e − 04	4.76e − 04	6.31e − 04	5.67e − 04	5.03e − 04	4.62e − 04	9.85e − 04
Distributed greedy	3.40e − 04	5.05e − 04	4.38e − 04	6.04e − 04	6.46e − 04	1.03e − 03	7.66e − 04	1.36e − 03	1.11e − 03	1.96e − 03
Hyperplane	2.02e − 04	2.81e − 04	3.63e − 04	4.16e − 04	4.68e − 04	5.15e − 04	3.79e − 04	4.77e − 04	4.55e − 04	9.05e − 04

Table A.6: Standard deviation of instatiation time in seconds for the five point hops last stencil, 32 processes per node and 3 dimensions

Reordering Scheme	Number of nodes									
	6	9	12	15	18	21	24	27	30	33
Cart no reorder	4.80e − 05	5.40e − 05	5.40e − 05	7.20e − 05	7.10e − 05	7.30e − 05	7.00e − 05	8.50e − 05	8.40e − 05	9.70e − 05
Cart reorder	6.00e − 06	7.00e − 06	6.00e − 06	8.00e − 06	8.00e − 06	9.00e − 06	1.30e − 05	8.00e − 06	1.30e − 05	7.40e − 05
Gropp	5.59e − 04	2.31e − 03	9.33e − 04	1.21e − 03	1.10e − 03	1.44e − 03	8.03e − 04	1.60e − 03	1.33e − 03	1.17e − 03
Central greedy	3.47e − 04	4.02e − 04	4.86e − 04	5.55e − 04	5.34e − 04	5.34e − 04	5.47e − 04	5.42e − 04	5.75e − 04	6.20e − 04
Distributed greedy	3.21e − 04	4.30e − 04	4.97e − 04	4.59e − 04	6.34e − 04	6.64e − 04	8.42e − 04	1.12e − 03	1.31e − 03	1.40e − 03
Hyperplane	2.30e − 04	2.95e − 04	4.22e − 04	4.17e − 04	4.96e − 04	5.42e − 04	3.88e − 04	5.37e − 04	5.35e − 04	4.46e − 04

Table A.7: Standard deviation of instatiation time in seconds for the component stencil, 32 processes per node and 3 dimensions

Reordering Scheme	**Number of nodes**									
	6	9	12	15	18	21	24	27	30	33
Cart no reorder	$4.40e-05$	$5.70e-05$	$5.50e-05$	$6.90e-05$	$7.10e-05$	$7.20e-05$	$7.10e-05$	$8.10e-05$	$8.00e-05$	$8.20e-05$
Cart reorder	$9.00e-06$	$6.00e-06$	$8.00e-06$	$9.00e-06$	$6.00e-06$	$1.20e-05$	$6.00e-06$	$1.10e-05$	$1.40e-05$	$1.50e-05$
Gropp	$5.55e-04$	$2.17e-03$	$1.06e-03$	$1.16e-03$	$1.72e-03$	$1.42e-03$	$7.79e-04$	$1.49e-03$	$1.50e-03$	$1.52e-03$
Central greedy	$9.20e-05$	$3.24e-04$	$6.55e-04$	$6.34e-04$	$6.42e-04$	$6.03e-04$	$4.98e-04$	$4.81e-04$	$3.57e-04$	$3.02e-04$
Distributed greedy	$2.25e-04$	$2.13e-04$	$5.19e-04$	$7.37e-04$	$9.95e-04$	$1.23e-03$	$1.67e-03$	$2.44e-03$	$3.61e-03$	$2.86e-03$
Hyperplane	$2.65e-04$	$2.99e-04$	$3.60e-04$	$4.32e-04$	$5.09e-04$	$7.09e-04$	$1.91e-04$	$8.56e-04$	$6.66e-04$	$7.55e-04$

A.2 Standard Deviation for Neighbor All-to-all Routine

Table A.8: Standard deviation for the average time bandwidth in bytes per seconds of the neighbor all-to-all routine, the five point stencil, 33 nodes, 32 processes per node and 3 dimensions

Number of Bytes	**Rank Reordering Scheme**					
	cart reorder	**dist greedy cart**	**nodecart**	**cart**	**central greedy cart**	**hyperplane cart**
80	$2.05e+06$	$2.05e+06$	$2.00e+06$	$2.00e+06$	$2.05e+06$	$2.05e+06$
160	$4.57e+06$	$4.44e+06$	$4.44e+06$	$4.44e+06$	$4.57e+06$	$4.44e+06$
240	$6.49e+06$	$6.49e+06$	$6.49e+06$	$6.67e+06$	$6.67e+06$	$6.67e+06$
320	$8.42e+06$	$8.42e+06$	$8.42e+06$	$8.42e+06$	$8.65e+06$	$8.42e+06$
400	$9.76e+06$	$9.76e+06$	$1.03e+07$	$1.03e+07$	$1.05e+07$	$1.03e+07$
480	$1.12e+07$	$1.14e+07$	$1.20e+07$	$1.20e+07$	$1.20e+07$	$1.17e+07$
560	$1.27e+07$	$1.27e+07$	$1.37e+07$	$1.37e+07$	$1.37e+07$	$1.33e+07$
640	$1.39e+07$	$1.39e+07$	$1.49e+07$	$1.52e+07$	$1.56e+07$	$1.49e+07$
800	$1.54e+07$	$1.54e+07$	$1.74e+07$	$1.74e+07$	$1.78e+07$	$1.67e+07$
1600	$2.16e+07$	$2.16e+07$	$2.67e+07$	$2.58e+07$	$2.76e+07$	$2.32e+07$
2400	$2.45e+07$	$2.45e+07$	$3.12e+07$	$2.93e+07$	$3.24e+07$	$2.61e+07$
3200	$2.76e+07$	$2.76e+07$	$3.40e+07$	$3.14e+07$	$3.76e+07$	$2.91e+07$
4000	$2.94e+07$	$2.92e+07$	$3.54e+07$	$3.20e+07$	$3.96e+07$	$3.08e+07$
4800	$2.96e+07$	$2.96e+07$	$3.50e+07$	$3.20e+07$	$4.03e+07$	$3.12e+07$
5600	$2.98e+07$	$2.98e+07$	$3.48e+07$	$3.20e+07$	$4.06e+07$	$3.13e+07$
6400	$3.00e+07$	$3.02e+07$	$3.56e+07$	$3.25e+07$	$4.13e+07$	$3.20e+07$
8000	$3.04e+07$	$3.05e+07$	$3.56e+07$	$3.28e+07$	$4.21e+07$	$3.25e+07$
16000	$3.07e+07$	$3.07e+07$	$3.63e+07$	$3.35e+07$	$4.27e+07$	$3.27e+07$
24000	$5.17e+07$	$5.22e+07$	$4.44e+07$	$3.88e+07$	$6.88e+07$	$5.29e+07$
32000	$5.51e+07$	$5.50e+07$	$4.64e+07$	$3.99e+07$	$7.69e+07$	$5.57e+07$
40000	$5.71e+07$	$5.69e+07$	$4.71e+07$	$4.02e+07$	$8.13e+07$	$5.78e+07$
48000	$5.84e+07$	$5.83e+07$	$4.81e+07$	$4.07e+07$	$8.32e+07$	$5.89e+07$
56000	$5.95e+07$	$5.95e+07$	$4.82e+07$	$4.04e+07$	$8.42e+07$	$6.00e+07$
64000	$6.02e+07$	$6.02e+07$	$4.86e+07$	$4.07e+07$	$8.51e+07$	$6.07e+07$
72000	$5.78e+07$	$5.79e+07$	$4.76e+07$	$4.02e+07$	$8.23e+07$	$5.96e+07$
80000	$5.82e+07$	$5.83e+07$	$4.76e+07$	$4.01e+07$	$8.32e+07$	$6.03e+07$
88000	$5.88e+07$	$5.87e+07$	$4.75e+07$	$4.02e+07$	$8.38e+07$	$6.11e+07$
96000	$5.90e+07$	$5.90e+07$	$4.74e+07$	$3.99e+07$	$8.40e+07$	$6.14e+07$
104000	$5.95e+07$	$5.94e+07$	$4.74e+07$	$4.00e+07$	$8.45e+07$	$6.19e+07$

Table A.9: Standard deviation for the average time bandwidth in bytes per seconds of the neighbor all-to-all routine, the nine point stencil, 33 nodes, 32 processes per node and 3 dimensions

Number of Bytes	**Rank Reordering Scheme**					
	cart reorder	**dist greedy cart**	**nodecart**	**cart**	**central greedy cart**	**hyperplane cart**
80	$6.15e+05$	$6.15e+05$	$5.93e+05$	$5.26e+05$	$5.52e+05$	$6.20e+05$
160	$1.30e+06$	$1.30e+06$	$1.28e+06$	$1.17e+06$	$1.23e+06$	$1.31e+06$
240	$1.88e+06$	$1.88e+06$	$1.88e+06$	$1.71e+06$	$1.79e+06$	$1.92e+06$
320	$2.37e+06$	$2.37e+06$	$2.41e+06$	$2.22e+06$	$2.29e+06$	$2.46e+06$
400	$2.65e+06$	$2.65e+06$	$2.76e+06$	$2.61e+06$	$2.65e+06$	$2.76e+06$
480	$2.96e+06$	$2.98e+06$	$3.16e+06$	$3.06e+06$	$3.04e+06$	$3.04e+06$
560	$3.20e+06$	$3.20e+06$	$3.48e+06$	$3.44e+06$	$3.35e+06$	$3.24e+06$
640	$3.35e+06$	$3.35e+06$	$3.74e+06$	$3.81e+06$	$3.64e+06$	$3.40e+06$
800	$3.52e+06$	$3.52e+06$	$3.96e+06$	$4.28e+06$	$3.90e+06$	$3.56e+06$
1600	$3.93e+06$	$3.93e+06$	$4.61e+06$	$5.35e+06$	$4.60e+06$	$3.95e+06$
2400	$4.02e+06$	$4.01e+06$	$4.75e+06$	$5.67e+06$	$4.76e+06$	$4.03e+06$
3200	$4.12e+06$	$4.12e+06$	$4.89e+06$	$5.88e+06$	$4.89e+06$	$4.10e+06$
4000	$4.20e+06$	$4.21e+06$	$4.90e+06$	$5.87e+06$	$4.97e+06$	$4.15e+06$
4800	$4.26e+06$	$4.26e+06$	$5.01e+06$	$6.04e+06$	$5.07e+06$	$4.21e+06$
5600	$4.28e+06$	$4.29e+06$	$5.07e+06$	$6.16e+06$	$5.15e+06$	$4.26e+06$
6400	$4.37e+06$	$4.37e+06$	$5.14e+06$	$6.34e+06$	$5.27e+06$	$4.34e+06$
8000	$4.38e+06$	$4.38e+06$	$5.11e+06$	$6.37e+06$	$5.28e+06$	$4.35e+06$
16000	$4.43e+06$	$4.43e+06$	$5.04e+06$	$6.46e+06$	$5.31e+06$	$4.34e+06$
24000	$4.24e+06$	$4.24e+06$	$4.65e+06$	$5.78e+06$	$4.96e+06$	$4.16e+06$
32000	$4.24e+06$	$4.24e+06$	$4.66e+06$	$5.80e+06$	$5.03e+06$	$4.17e+06$
40000	$4.24e+06$	$4.23e+06$	$4.62e+06$	$5.75e+06$	$4.98e+06$	$4.17e+06$
48000	$4.25e+06$	$4.24e+06$	$4.64e+06$	$5.80e+06$	$5.03e+06$	$4.19e+06$
56000	$4.23e+06$	$4.22e+06$	$4.61e+06$	$5.73e+06$	$5.00e+06$	$4.17e+06$
64000	$4.23e+06$	$4.22e+06$	$4.61e+06$	$5.72e+06$	$5.01e+06$	$4.18e+06$
72000	$4.37e+06$	$4.37e+06$	$4.83e+06$	$6.01e+06$	$5.29e+06$	$4.26e+06$
80000	$4.38e+06$	$4.36e+06$	$4.82e+06$	$6.00e+06$	$5.29e+06$	$4.26e+06$
88000	$4.38e+06$	$4.37e+06$	$4.81e+06$	$5.99e+06$	$5.28e+06$	$4.28e+06$
96000	$4.38e+06$	$4.37e+06$	$4.81e+06$	$5.99e+06$	$5.28e+06$	$4.28e+06$
104000	$4.38e+06$	$4.36e+06$	$4.80e+06$	$5.97e+06$	$5.27e+06$	$4.28e+06$

Table A.10: Standard deviation for the average time bandwidth in bytes per seconds of the neighbor all-to-all routine, the component stencil, 33 nodes, 32 processes per node and 3 dimensions

Number of Bytes	**Rank Reordering Scheme**					
	cart reorder	**dist greedy cart**	**nodecart**	**cart**	**central greedy cart**	**hyperplane cart**
80	$3.48e+06$	$3.48e+06$	$2.76e+06$	$2.76e+06$	$3.20e+06$	$3.33e+06$
160	$7.62e+06$	$7.62e+06$	$6.15e+06$	$6.15e+06$	$7.27e+06$	$7.27e+06$
240	$1.09e+07$	$1.14e+07$	$8.89e+06$	$8.89e+06$	$1.04e+07$	$1.09e+07$
320	$1.45e+07$	$1.45e+07$	$1.14e+07$	$1.14e+07$	$1.23e+07$	$1.45e+07$
400	$1.67e+07$	$1.74e+07$	$1.38e+07$	$1.38e+07$	$1.67e+07$	$1.74e+07$
480	$1.66e+07$	$1.92e+07$	$1.55e+07$	$1.60e+07$	$2.00e+07$	$1.66e+07$
560	$2.07e+07$	$2.15e+07$	$1.81e+07$	$1.75e+07$	$2.24e+07$	$2.15e+07$
640	$2.37e+07$	$2.37e+07$	$2.00e+07$	$2.00e+07$	$2.46e+07$	$2.46e+07$
800	$2.67e+07$	$2.67e+07$	$2.29e+07$	$2.29e+07$	$2.96e+07$	$2.76e+07$
1600	$3.40e+07$	$3.40e+07$	$3.64e+07$	$3.72e+07$	$4.44e+07$	$3.56e+07$
2400	$3.64e+07$	$3.64e+07$	$4.53e+07$	$4.71e+07$	$5.22e+07$	$3.81e+07$
3200	$3.81e+07$	$3.81e+07$	$5.52e+07$	$5.61e+07$	$5.82e+07$	$3.95e+07$
4000	$3.92e+07$	$3.92e+07$	$6.15e+07$	$6.15e+07$	$5.97e+07$	$4.04e+07$
4800	$4.03e+07$	$4.00e+07$	$6.49e+07$	$6.32e+07$	$6.32e+07$	$4.10e+07$
5600	$4.09e+07$	$4.09e+07$	$6.59e+07$	$6.36e+07$	$6.51e+07$	$4.15e+07$
6400	$4.16e+07$	$4.16e+07$	$6.81e+07$	$6.60e+07$	$6.74e+07$	$4.24e+07$
8000	$4.23e+07$	$4.21e+07$	$7.02e+07$	$6.72e+07$	$6.96e+07$	$4.32e+07$
16000	$4.48e+07$	$4.47e+07$	$7.11e+07$	$6.84e+07$	$7.51e+07$	$4.76e+07$
24000	$5.26e+07$	$5.25e+07$	$1.12e+08$	$1.01e+08$	$9.41e+07$	$5.32e+07$
32000	$5.50e+07$	$5.49e+07$	$1.32e+08$	$1.14e+08$	$1.02e+08$	$5.57e+07$
40000	$5.71e+07$	$5.68e+07$	$1.51e+08$	$1.25e+08$	$1.09e+08$	$5.78e+07$
48000	$5.84e+07$	$5.81e+07$	$1.62e+08$	$1.31e+08$	$1.11e+08$	$5.88e+07$
56000	$5.96e+07$	$5.94e+07$	$1.70e+08$	$1.36e+08$	$1.14e+08$	$5.98e+07$
64000	$6.02e+07$	$6.02e+07$	$1.76e+08$	$1.40e+08$	$1.16e+08$	$6.06e+07$
72000	$5.77e+07$	$5.78e+07$	$1.70e+08$	$1.36e+08$	$1.13e+08$	$6.00e+07$
80000	$5.81e+07$	$5.80e+07$	$1.73e+08$	$1.37e+08$	$1.14e+08$	$6.06e+07$
88000	$5.83e+07$	$5.84e+07$	$1.74e+08$	$1.38e+08$	$1.15e+08$	$6.09e+07$
96000	$5.85e+07$	$5.86e+07$	$1.75e+08$	$1.39e+08$	$1.15e+08$	$6.13e+07$
104000	$5.87e+07$	$5.89e+07$	$1.77e+08$	$1.39e+08$	$1.16e+08$	$6.15e+07$

Table A.11: Standard deviation for the average time bandwidth in bytes per seconds of the neighbor all-to-all routine, the diagonal stencil, 33 nodes, 32 processes per node and 3 dimensions

Number of Bytes	**Rank Reordering Scheme**					
	cart reorder	**dist greedy cart**	**nodecart**	**cart**	**central greedy cart**	**hyperplane cart**
80	$2.22e+06$	$2.22e+06$	$1.63e+06$	$1.67e+06$	$1.74e+06$	$2.22e+06$
160	$3.81e+06$	$3.81e+06$	$3.64e+06$	$3.64e+06$	$3.72e+06$	$3.81e+06$
240	$4.62e+06$	$4.62e+06$	$5.33e+06$	$5.45e+06$	$5.33e+06$	$4.62e+06$
320	$5.25e+06$	$5.25e+06$	$6.96e+06$	$6.96e+06$	$6.67e+06$	$5.16e+06$
400	$5.56e+06$	$5.56e+06$	$8.33e+06$	$8.51e+06$	$7.55e+06$	$5.48e+06$
480	$5.85e+06$	$5.85e+06$	$9.80e+06$	$9.80e+06$	$8.28e+06$	$5.78e+06$
560	$6.09e+06$	$6.09e+06$	$1.12e+07$	$1.12e+07$	$8.89e+06$	$5.96e+06$
640	$6.27e+06$	$6.27e+06$	$1.25e+07$	$1.25e+07$	$9.28e+06$	$6.15e+06$
800	$6.50e+06$	$6.50e+06$	$1.45e+07$	$1.45e+07$	$9.88e+06$	$6.35e+06$
1600	$6.99e+06$	$6.99e+06$	$2.25e+07$	$2.25e+07$	$1.11e+07$	$6.84e+06$
2400	$7.08e+06$	$7.08e+06$	$2.58e+07$	$2.58e+07$	$1.14e+07$	$6.92e+06$
3200	$7.10e+06$	$7.10e+06$	$2.81e+07$	$2.78e+07$	$1.16e+07$	$6.96e+06$
4000	$7.09e+06$	$7.10e+06$	$2.90e+07$	$2.86e+07$	$1.17e+07$	$6.98e+06$
4800	$7.12e+06$	$7.12e+06$	$2.87e+07$	$2.84e+07$	$1.17e+07$	$6.98e+06$
5600	$7.29e+06$	$7.29e+06$	$2.87e+07$	$2.83e+07$	$1.17e+07$	$7.13e+06$
6400	$7.48e+06$	$7.48e+06$	$2.95e+07$	$2.91e+07$	$1.19e+07$	$7.28e+06$
8000	$7.51e+06$	$7.51e+06$	$2.99e+07$	$2.93e+07$	$1.20e+07$	$7.33e+06$
16000	$7.85e+06$	$7.86e+06$	$3.11e+07$	$3.04e+07$	$1.25e+07$	$7.55e+06$
24000	$7.56e+06$	$7.57e+06$	$3.31e+07$	$3.37e+07$	$1.21e+07$	$7.25e+06$
32000	$7.60e+06$	$7.64e+06$	$3.41e+07$	$3.46e+07$	$1.24e+07$	$7.37e+06$
40000	$7.61e+06$	$7.64e+06$	$3.43e+07$	$3.49e+07$	$1.24e+07$	$7.32e+06$
48000	$7.68e+06$	$7.71e+06$	$3.50e+07$	$3.56e+07$	$1.25e+07$	$7.39e+06$
56000	$7.66e+06$	$7.69e+06$	$3.49e+07$	$3.54e+07$	$1.24e+07$	$7.36e+06$
64000	$7.67e+06$	$7.71e+06$	$3.53e+07$	$3.56e+07$	$1.25e+07$	$7.40e+06$
72000	$7.71e+06$	$7.72e+06$	$3.59e+07$	$3.57e+07$	$1.26e+07$	$7.34e+06$
80000	$7.72e+06$	$7.72e+06$	$3.59e+07$	$3.58e+07$	$1.26e+07$	$7.36e+06$
88000	$7.71e+06$	$7.73e+06$	$3.58e+07$	$3.58e+07$	$1.26e+07$	$7.35e+06$
96000	$7.72e+06$	$7.73e+06$	$3.57e+07$	$3.56e+07$	$1.26e+07$	$7.35e+06$
104000	$7.71e+06$	$7.73e+06$	$3.56e+07$	$3.57e+07$	$1.26e+07$	$7.35e+06$

Table A.12: Standard deviation for the average time bandwidth in bytes per seconds of the neighbor all-to-all routine, the crank nicolson stencil, 33 nodes, 32 processes per node and 3 dimensions

Number of Bytes	**Rank Reordering Scheme**					
	cart reorder	**dist greedy cart**	**nodecart**	**cart**	**central greedy cart**	**hyperplane cart**
80	1.57e+06	1.57e+06	1.43e+06	1.51e+06	1.45e+06	1.54e+06
160	3.48e+06	3.48e+06	3.20e+06	3.33e+06	3.27e+06	3.33e+06
240	5.00e+06	5.00e+06	4.71e+06	4.90e+06	4.80e+06	5.00e+06
320	6.40e+06	6.40e+06	6.04e+06	6.40e+06	6.15e+06	6.53e+06
400	7.55e+06	7.55e+06	7.27e+06	7.55e+06	7.41e+06	7.69e+06
480	8.57e+06	8.57e+06	8.42e+06	8.89e+06	8.57e+06	8.89e+06
560	9.49e+06	9.66e+06	9.66e+06	1.00e+07	9.82e+06	1.00e+07
640	1.05e+07	1.05e+07	1.07e+07	1.10e+07	1.08e+07	1.10e+07
800	1.14e+07	1.14e+07	1.21e+07	1.23e+07	1.25e+07	1.21e+07
1600	1.39e+07	1.39e+07	1.63e+07	1.57e+07	1.74e+07	1.48e+07
2400	1.45e+07	1.45e+07	1.75e+07	1.66e+07	1.92e+07	1.54e+07
3200	1.50e+07	1.50e+07	1.82e+07	1.71e+07	2.08e+07	1.58e+07
4000	1.53e+07	1.52e+07	1.84e+07	1.72e+07	2.15e+07	1.60e+07
4800	1.53e+07	1.53e+07	1.85e+07	1.73e+07	2.14e+07	1.61e+07
5600	1.54e+07	1.54e+07	1.87e+07	1.74e+07	2.15e+07	1.61e+07
6400	1.56e+07	1.56e+07	1.87e+07	1.75e+07	2.18e+07	1.63e+07
8000	1.57e+07	1.57e+07	1.88e+07	1.75e+07	2.21e+07	1.64e+07
16000	1.62e+07	1.62e+07	1.97e+07	1.85e+07	2.31e+07	1.68e+07
24000	1.83e+07	1.82e+07	1.98e+07	1.85e+07	2.71e+07	1.78e+07
32000	1.92e+07	1.92e+07	2.02e+07	1.87e+07	2.89e+07	1.87e+07
40000	1.92e+07	1.91e+07	2.01e+07	1.87e+07	2.95e+07	1.92e+07
48000	1.94e+07	1.93e+07	2.04e+07	1.88e+07	3.04e+07	1.95e+07
56000	1.91e+07	1.90e+07	2.03e+07	1.87e+07	3.07e+07	1.91e+07
64000	1.93e+07	1.92e+07	2.04e+07	1.87e+07	3.12e+07	1.94e+07
72000	1.99e+07	1.99e+07	2.05e+07	1.89e+07	3.04e+07	2.11e+07
80000	1.99e+07	1.99e+07	2.05e+07	1.89e+07	3.05e+07	2.13e+07
88000	2.00e+07	2.00e+07	2.05e+07	1.89e+07	3.06e+07	2.13e+07
96000	2.00e+07	2.00e+07	2.04e+07	1.88e+07	3.06e+07	2.12e+07
104000	2.00e+07	2.00e+07	2.04e+07	1.88e+07	3.06e+07	2.13e+07

Table A.13: Standard deviation for the average time bandwidth in bytes per seconds of the neighbor all-to-all routine, the five point hops first stencil, 33 nodes, 32 processes per node and 3 dimensions

Number of Bytes	**Rank Reordering Scheme**					
	cart reorder	**dist greedy cart**	**nodecart**	**cart**	**central greedy cart**	**hyperplane cart**
80	1.48e + 06	1.48e + 06	1.29e + 06	1.29e + 06	1.43e + 06	1.45e + 06
160	3.14e + 06	3.14e + 06	2.91e + 06	2.91e + 06	3.27e + 06	3.14e + 06
240	4.62e + 06	4.62e + 06	4.21e + 06	4.14e + 06	4.80e + 06	4.62e + 06
320	6.04e + 06	6.04e + 06	5.42e + 06	5.52e + 06	6.27e + 06	6.04e + 06
400	6.90e + 06	7.02e + 06	6.45e + 06	6.56e + 06	7.27e + 06	7.14e + 06
480	8.00e + 06	8.00e + 06	7.62e + 06	7.62e + 06	8.73e + 06	8.42e + 06
560	9.03e + 06	8.89e + 06	8.62e + 06	8.62e + 06	9.82e + 06	9.49e + 06
640	9.85e + 06	9.85e + 06	9.70e + 06	9.70e + 06	1.10e + 07	1.05e + 07
800	1.07e + 07	1.07e + 07	1.11e + 07	1.11e + 07	1.29e + 07	1.19e + 07
1600	1.34e + 07	1.33e + 07	1.60e + 07	1.60e + 07	2.00e + 07	1.67e + 07
2400	1.41e + 07	1.41e + 07	1.76e + 07	1.80e + 07	2.40e + 07	1.82e + 07
3200	1.51e + 07	1.51e + 07	1.89e + 07	1.96e + 07	2.69e + 07	1.96e + 07
4000	1.54e + 07	1.54e + 07	1.94e + 07	2.03e + 07	2.86e + 07	2.01e + 07
4800	1.55e + 07	1.55e + 07	1.92e + 07	2.03e + 07	2.94e + 07	2.03e + 07
5600	1.56e + 07	1.56e + 07	1.96e + 07	2.03e + 07	2.95e + 07	2.03e + 07
6400	1.58e + 07	1.58e + 07	1.98e + 07	2.06e + 07	3.06e + 07	2.06e + 07
8000	1.58e + 07	1.59e + 07	1.95e + 07	2.04e + 07	3.12e + 07	2.09e + 07
16000	1.59e + 07	1.59e + 07	2.09e + 07	2.20e + 07	3.19e + 07	2.12e + 07
24000	2.14e + 07	2.14e + 07	2.17e + 07	2.39e + 07	4.63e + 07	2.88e + 07
32000	2.17e + 07	2.18e + 07	2.19e + 07	2.45e + 07	5.06e + 07	3.00e + 07
40000	2.16e + 07	2.17e + 07	2.16e + 07	2.46e + 07	5.31e + 07	3.05e + 07
48000	2.19e + 07	2.20e + 07	2.17e + 07	2.49e + 07	5.45e + 07	3.09e + 07
56000	2.17e + 07	2.18e + 07	2.13e + 07	2.47e + 07	5.54e + 07	3.08e + 07
64000	2.17e + 07	2.18e + 07	2.13e + 07	2.49e + 07	5.59e + 07	3.11e + 07
72000	2.17e + 07	2.16e + 07	2.11e + 07	2.46e + 07	5.39e + 07	3.13e + 07
80000	2.16e + 07	2.15e + 07	2.11e + 07	2.46e + 07	5.41e + 07	3.15e + 07
88000	2.16e + 07	2.15e + 07	2.09e + 07	2.45e + 07	5.42e + 07	3.15e + 07
96000	2.15e + 07	2.15e + 07	2.08e + 07	2.45e + 07	5.44e + 07	3.16e + 07
104000	2.15e + 07	2.14e + 07	2.08e + 07	2.44e + 07	5.44e + 07	3.16e + 07

Table A.14: Standard deviation for the average time bandwidth in bytes per seconds of the neighbor all-to-all routine, the five point hops last stencil, 33 nodes, 32 processes per node and 3 dimensions

Number of Bytes	**Rank Reordering Scheme**					
	cart reorder	**dist greedy cart**	**nodecart**	**cart**	**central greedy cart**	**hyperplane cart**
80	$1.27e+06$	$1.29e+06$	$1.31e+06$	$1.31e+06$	$1.29e+06$	$1.29e+06$
160	$2.91e+06$	$2.86e+06$	$2.91e+06$	$2.91e+06$	$2.86e+06$	$2.86e+06$
240	$4.21e+06$	$4.21e+06$	$4.29e+06$	$4.21e+06$	$4.14e+06$	$4.21e+06$
320	$5.42e+06$	$5.42e+06$	$5.52e+06$	$5.52e+06$	$5.52e+06$	$5.52e+06$
400	$6.06e+06$	$6.45e+06$	$6.56e+06$	$6.56e+06$	$6.56e+06$	$6.56e+06$
480	$7.38e+06$	$7.38e+06$	$7.62e+06$	$7.62e+06$	$7.62e+06$	$7.62e+06$
560	$8.36e+06$	$8.36e+06$	$8.62e+06$	$8.62e+06$	$8.62e+06$	$8.62e+06$
640	$9.28e+06$	$9.14e+06$	$9.70e+06$	$9.55e+06$	$9.55e+06$	$9.55e+06$
800	$1.04e+07$	$1.04e+07$	$1.11e+07$	$1.07e+07$	$1.11e+07$	$1.10e+07$
1600	$1.50e+07$	$1.50e+07$	$1.65e+07$	$1.63e+07$	$1.63e+07$	$1.58e+07$
2400	$1.74e+07$	$1.75e+07$	$1.86e+07$	$1.88e+07$	$1.94e+07$	$1.85e+07$
3200	$1.99e+07$	$2.04e+07$	$1.99e+07$	$2.08e+07$	$2.25e+07$	$2.13e+07$
4000	$2.19e+07$	$2.20e+07$	$2.02e+07$	$2.14e+07$	$2.45e+07$	$2.30e+07$
4800	$2.26e+07$	$2.27e+07$	$2.02e+07$	$2.13e+07$	$2.54e+07$	$2.34e+07$
5600	$2.31e+07$	$2.31e+07$	$2.05e+07$	$2.18e+07$	$2.57e+07$	$2.39e+07$
6400	$2.37e+07$	$2.37e+07$	$2.08e+07$	$2.21e+07$	$2.64e+07$	$2.46e+07$
8000	$2.41e+07$	$2.41e+07$	$2.09e+07$	$2.22e+07$	$2.70e+07$	$2.52e+07$
16000	$2.41e+07$	$2.42e+07$	$2.22e+07$	$2.32e+07$	$2.71e+07$	$2.54e+07$
24000	$4.90e+07$	$4.91e+07$	$2.34e+07$	$2.63e+07$	$5.05e+07$	$4.97e+07$
32000	$5.39e+07$	$5.38e+07$	$2.40e+07$	$2.70e+07$	$5.91e+07$	$5.51e+07$
40000	$5.66e+07$	$5.64e+07$	$2.40e+07$	$2.70e+07$	$6.37e+07$	$5.76e+07$
48000	$5.77e+07$	$5.75e+07$	$2.43e+07$	$2.74e+07$	$6.54e+07$	$5.85e+07$
56000	$5.93e+07$	$5.83e+07$	$2.42e+07$	$2.73e+07$	$6.73e+07$	$5.96e+07$
64000	$5.99e+07$	$5.99e+07$	$2.43e+07$	$2.75e+07$	$6.83e+07$	$6.02e+07$
72000	$5.65e+07$	$5.68e+07$	$2.42e+07$	$2.68e+07$	$6.48e+07$	$6.09e+07$
80000	$5.73e+07$	$5.74e+07$	$2.42e+07$	$2.68e+07$	$6.55e+07$	$6.14e+07$
88000	$5.76e+07$	$5.77e+07$	$2.42e+07$	$2.68e+07$	$6.57e+07$	$6.17e+07$
96000	$5.76e+07$	$5.78e+07$	$2.41e+07$	$2.67e+07$	$6.56e+07$	$6.17e+07$
104000	$5.80e+07$	$5.82e+07$	$2.41e+07$	$2.67e+07$	$6.59e+07$	$6.20e+07$

Bibliography

[1] G. Baolai. "Parallel Numerical Solution of PDEs with Message Passing". The University of Western Ontario. 2008.

[2] S. Beamer, A. Buluç, K. Asanovic, and D. Patterson. "Distributed Memory Breadth-First Search Revisited: Enabling Bottom-Up Search". In: *2013 IEEE International Symposium on Parallel Distributed Processing, Workshops and Phd Forum.* 2013, pp. 1618–1627. DOI: `10.1109/IPDPSW.2013.159`.

[3] A. Bhatele and L. V. Kalé. "Benefits of Topology Aware Mapping for Mesh Interconnects". In: *Parallel Processing Letters* 18 (2008), pp. 549–566. DOI: `10.1142/S0129626408003569`.

[4] S. H. Bokhari. "On the Mapping Problem". In: *IEEE Trans. Comput.* 30.3 (1981). DOI: `10.1109/TC.1981.1675756`.

[5] B. Brandfass, T. Alrutz, and T. Gerhold. "Rank reordering for MPI communication optimization". In: *Computers & Fluids* 80 (2013), pp. 372–380. DOI: `10.1016/j.compfluid.2012.01.019`.

[6] G. S. Brodal, G. Lagogiannis, and R. E. Tarjan. "Strict Fibonacci Heaps". In: *Proceedings of the Forty-fourth Annual ACM Symposium on Theory of Computing.* 2012, pp. 1177–1184. DOI: `10.1145/2213977.2214082`.

[7] F. Broquedis, J. Clet-Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst. "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications". In: *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing.* 2010. DOI: `10.1109/PDP.2010.67`.

[8] C. W. Commander. "A survey of the quadratic assignment problem, with applications". In: *Morehead Electronic Journal of Applicable Mathematics* 4 (2005).

[9] J. Crank and P. Nicolson. “A practical method for numerical evaluation of solutions of partial differential equations of the heat-conduction type”. In: *Mathematical Proceedings of the Cambridge Philosophical Society* 43 (1947), pp. 50–67. DOI: 10.1017/S0305004100023197.

[10] Y. Cui, R. Moore, K. Olsen, A. Chourasia, P. Maechling, B. Minster, S. Day, Y. Hu, J. Zhu, and T. Jordan. “Toward petascale earthquake simulations”. In: *Acta Geotechnica* 4 (2009), pp. 79–93. DOI: 10.1007/s11440-008-0055-2.

[11] R. D. Falgout and U. M. Yang. “hypre: A Library of High Performance Preconditioners”. In: *Computational Science — ICCS 2002*. 2002, pp. 632–641. ISBN: 978-3-540-47789-1.

[12] C. M. Fiduccia and R. M. Mattheyses. “A Linear-Time Heuristic for Improving Network Partitions”. In: *19th Design Automation Conference*. 1982, pp. 175–181. DOI: 10.1109/DAC.1982.1585498.

[13] W. D. Gropp. “Using Node and Socket Information to Implement MPI Cartesian Topologies”. In: *Parallel Computing* 85 (2019), pp. 98–108. DOI: 10.1016/j.parco.2019.01.001.

[14] W. D. Gropp. “Using Node Information to Implement MPI Cartesian Topologies”. In: *25th European MPI Users’ Group Meeting (EuroMPI)*. 2018. DOI: 10.1145/3236367.3236377.

[15] T. Hatazaki. “Rank Reordering Strategy for MPI Topology Creation Functions”. In: *Proceedings of the 5th European PVM/MPI Users’ Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*. 1998, pp. 188–195. ISBN: 3-540-65041-5.

[16] J. C. Hayes, M. L. Norman, R. A. Fiedler, J. O. Bordner, P. S. Li, S. E. Clark, A. ud-Doula, and M.-M. Mac Low. “Simulating Radiating and Magnetized Flows in Multiple Dimensions with ZEUS-MP”. In: *The Astrophysical Journal Supplement Series* 165 (2006), pp. 188–228. DOI: 10.1086/504594.

[17] C. H. Heider. *A Computationally Simplified Pair-Exchange Algorithm for the Quadratic Assignment Problem*. Tech. rep. Center for Naval Analysis AD 756 503, 1972.

[18] E. Jeannot and G. Mercier. “Near-optimal Placement of MPI Processes on Hierarchical NUMA Architectures”. In: *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part II (Euro-Par)*. 2010, pp. 199–210. ISBN: 3-642-15290-2, 978-3-642-15290-0.

[19] L. V. Kalé, E. Bohm, C. Mendes, T. Wilmarth, and G. Zheng. "Programming petascale applications with Charm++ and AMPI". In: *Petascale Computing: Algorithms and Applications* (2007), pp. 421–441. DOI: 10.1201/9781584889106.ch20.

[20] L. V. Kalé and S. Krishnan. "Charm++: Parallel programming with message-driven objects". In: *Parallel programming using C++* (1996), pp. 175–213. DOI: 10.1145/165854.165874.

[21] B. W. Kernighan and S. Lin. "An efficient heuristic procedure for partitioning graphs". In: *The Bell System Technical Journal* 49.2 (1970), pp. 291–307. DOI: 10.1002/j.1538-7305.1970.tb01770.x.

[22] T. Koopmans and M. J. Beckmann. *Assignment Problems and the Location of Economic Activities.* Cowles Foundation Discussion Papers 4. Cowles Foundation for Research in Economics, Yale University, 1955. DOI: 10.2307/1907742.

[23] C. E. Leiserson and T. B. Schardl. "A Work-efficient Parallel Breadth-first Search Algorithm (or How to Cope with the Nondeterminism of Reducers)". In: *Proceedings of the Twenty-second Annual ACM Symposium on Parallelism in Algorithms and Architectures.* SPAA '10. 2010, pp. 303–314. DOI: 10.1145/1810479.1810534.

[24] A. Mamidala, L. Chai, H.-W. Jin, and D. Panda. "Efficient SMP-aware MPI-level broadcast over InfiniBand's hardware multicast". In: *Proceedings 20th IEEE International Parallel Distributed Processing Symposium.* 2006, 8 pp. DOI: 10.1109/IPDPS.2006.1639562.

[25] G. Mercier and E. Jeannot. "Improving MPI Applications Performance on Multicore Clusters with Rank Reordering". In: *Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, (EuroMPI).* 2011, pp. 39–49. DOI: 10.1007/978-3-642-24449-0_7.

[26] S. H. Mirsadeghi and A. Afsahi. "Topology-Aware Rank Reordering for MPI Collectives". In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW).* 2016, pp. 1759–1768. DOI: 10.1109/IPDPSW.2016.139.

[27] MPI Forum. *MPI: A Message-Passing Interface Standard. Version 3.1.* 2015. URL: www.mpi-forum.org.

[28] H. Müller-Merbach. "Optimale Reihenfolgen innerhalb mathematischer Algorithmen". In: *Optimale Reihenfolgen.* 1970, pp. 205–212. DOI: 10.1007/978-3-642-87727-8_10.

[29] C. Niethammer and R. Rabenseifner. “An MPI Interface for Application and Hardware Aware Cartesian Topology Optimization”. In: *Proceedings of the 26th European MPI Users' Group Meeting (EuroMPI)*. 2019. DOI: 10.1145/3343211.3343217.

[30] *Open Fabrics Enterprise Distribution*. URL: http://www.openfabrics.org/.

[31] F. Pellegrini and J. Roman. “Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs”. In: *High-Performance Computing and Networking*. 1996, pp. 493–498. ISBN: 978-3-540-49955-8.

[32] N. Saitou and M. Nei. “The neighbor-joining method: a new method for reconstructing phylogenetic trees”. In: *Mol Biol Evol* 4 (1987), pp. 406–425. DOI: 10.1093/oxfordjournals.molbev.a040454.

[33] C. Schulz and J. L. Träff. “Better Process Mapping and Sparse Quadratic Assignment”. In: *16th International Symposium on Experimental Algorithms (SEA)*. 2017. DOI: 10.4230/LIPIcs.SEA.2017.4.

[34] C. Schulz and J. L. Träff. “VieM v1.00 - Vienna Mapping and Sparse Quadratic Assignment User Guide”. In: *CoRR* abs/1703.05509 (2017). arXiv: 1703.05509.

[35] D. Schwamborn, T. Gerhold, and R. Heinrich. “The DLR TAU-Code: Recent Applications in Research and Industry”. In: *ECCOMAS CFD 2006 CONFERENCE*. 2006. URL: https://elib.dlr.de/22421/.

[36] H. Subramoni, S. Potluri, K. Kandalla, B. Barth, J. Vienne, J. Keasler, K. Tomko, K. Schulz, A. Moody, and D. K. Panda. “Design of a scalable InfiniBand topology service to enable network-topology-aware placement of processes”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. 2012, pp. 1–12. DOI: 10.1109/SC.2012.47.

[37] N. Sweilam, H. Moharram, and S. Ahmed. “On the parallel iterative finite difference algorithm for 2-D Poisson's equation with MPI cluster”. In: *Proceedings of the 8th International Conference on Informatics and Systems (INFOS)*. 2012, pp. MM–78.

[38] R. Thakur, R. Rabenseifner, and W. D. Gropp. “Optimization of Collective Communication Operations in MPICH”. In: *International Journal of High Performance Computation Applications* (2005), pp. 49–66. DOI: 10.1177/1094342005051521.

[39] *The OSU Micro-Benchmarks.* URL: `http://mvapich.cse.ohio-state.edu/benchmarks/`.

[40] J. L. Träff. “Implementing the MPI Process Topology Mechanism”. In: *SC ’02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*. 2002. DOI: `10.1109/SC.2002.10045`.

[41] J. L. Träff. “Direct graph k-partitioning with a Kernighan–Lin like heuristic”. In: *Operations Research Letters* 34.6 (2006), pp. 621–629. DOI: `https://doi.org/10.1016/j.orl.2005.10.003`.

[42] J. L. Träff, F. D. Lübbe, A. Rougier, and S. Hunold. “Isomorphic, Sparse MPI-like Collective Communication Operations for Parallel Stencil Computations”. In: *22nd European MPI Users’ Group Meeting (EuroMPI)*. 2015. DOI: `10.1145/2802658.2802663`.

[43] A. N. H. Zaied and L. A. E.-F. Shawky. “A Survey of Quadratic Assignment Problems”. In: *International Journal of Computer Applications* 101.6 (2014), pp. 28–36. DOI: `10.5120/17693-8662`.