Efficient Process-to-Node Mapping Algorithms for Stencil Computations

Konrad von Kirchbach
TU Wien, Faculty of Informatics
Vienna, Austria
kirchbach@par.tuwien.ac.at

Markus Lehr
TU Wien, Faculty of Informatics
Vienna, Austria
lehr@par.tuwien.ac.at

Sascha Hunold
TU Wien, Faculty of Informatics
Vienna, Austria
hunold@par.tuwien.ac.at

Christian Schulz
University of Vienna, Faculty of Computer Science
Vienna, Austria
christian.schulz@univie.ac.at

Jesper Larsson Träff
TU Wien, Faculty of Informatics
Vienna, Austria
traff@par.tuwien.ac.at

***Abstract*—Good process-to-compute-node mappings can be decisive for well performing HPC applications. A special, important class of process-to-node mapping problems is the problem of mapping processes that communicate in a sparse stencil pattern to Cartesian grids. By thoroughly exploiting the inherently present structure in this type of problem, we devise three novel distributed algorithms that are able to handle arbitrary stencil communication patterns effectively. We analyze the expected performance of our algorithms based on an abstract model of inter- and intra-node communication. An extensive experimental evaluation on several HPC machines shows that our algorithms are up to two orders of magnitude faster in running time than a (sequential) high-quality general graph mapping tool, while obtaining similar results in communication performance. Furthermore, our algorithms also achieve significantly better mapping quality compared to previous state-of-the-art Cartesian grid mapping algorithms. This results in up to a threefold performance improvement of an `MPI_Neighbor_alltoall` exchange operation. Our new algorithms can be used to implement the `MPI_Cart_create` functionality.**

***Index Terms*—MPI, Process Mapping, Stencil Computations**

I. Introduction

The communication performance of applications running on High-Performance Computing (HPC) systems depends on a variety of factors like the capability and topology of the underlying communication system, the required communication (patterns, frequencies, volumes, and dependencies) between processes, and the software and algorithms used to realize the communication. If the communication pattern is known, and if a hardware topology description is given, it is natural to attempt to find a good mapping of the application processes onto the hardware processors such that pairs of processes that frequently communicate large amounts of data become located closely, see Figure 1.

Many important scientific computing applications involve stencil computations. For example, stencil computations are used for climate and ocean modeling [1], in computational electromagnetic codes [2], [3], for image-processing [4], in Jacobi or multigrid solvers [5], for earthquake simulations [6] or in general in simulations systems such as OpenLB [7].

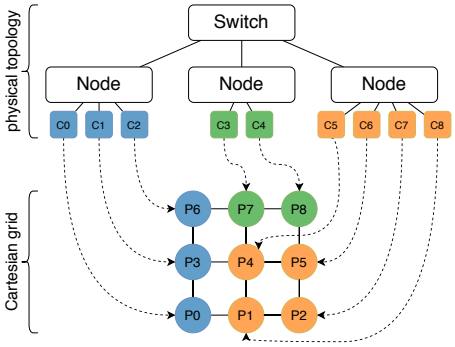


Figure 1: Motivational example: Given a set of compute nodes with possibly different numbers of processes per node, and a computational grid, find a mapping of the processes on the nodes to the grid, s.t. the number of communication edges between compute nodes is minimized.

In most cases, elements of a d-dimensional matrix are repeatedly updated using the values of fixed *stencil* pattern of neighboring elements. When run on a parallel computer, this yields communication patterns that are very regular and more or less symmetric depending on the organization of the processors. More precisely, each processing element exchanges data repeatedly with a small set of neighboring processing elements, and all processing element neighborhoods have the same structure. In this situation, in each exchange step, all processes communicate with other processes and all follow the same pattern determined by the computational stencil and the organization of the processes.

The Message Passing Interface (MPI) [8] supports complex communication patterns by providing functions to specify virtual process topologies by process neighborhoods. Using for instance Cartesian topologies the user can refer to processes by rank or by coordinate vectors. Moreover, MPI supports neighborhood collective operations such as `MPI_Neighbor_alltoall` which make it possible for the MPI library to exploit (regular) communication patterns

978-1-7281-6677-3/20/$31.00

to provide more efficient data exchange operations. MPI also defines functionality to reorder processes in order to optimize the communication performance, however, at the moment most MPI libraries do not actually perform such remapping (in the general case).

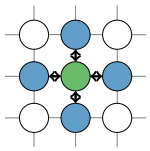
Contribution. We make the following contributions:

- We show that the general Cartesian mapping problem under stencil patterns is NP-hard. This result motivates our work on heuristic algorithms for the problem.
- We present new algorithms for the process-mapping problem for stencil patterns which in contrast to previous solutions are also applicable to cases where 1) the number of `MPI` processes per node is different and 2) where the number of processes is not factorizable or divisible by the number of processes per node, and 3) consider the case of arbitrary stencil patterns, not only the nearest-neighbor stencils implied by the `MPI` specification.
- We perform an extensive experimental evaluation and benchmark the time needed for an `MPI_Neighbor_alltoall` operation. The results show that our algorithms significantly outperform previous solutions in terms of communication performance as well as initialization time. For example on our biggest tested instance, our algorithms are up to two orders of magnitude faster in running time than the (sequential) high-quality general graph mapping tool Vienna Mapping (`VieM`), while obtaining similar results in communication performance. Moreover, our algorithms are up to three times faster than other Cartesian grid mapping algorithms, and achieving significantly better mapping quality.

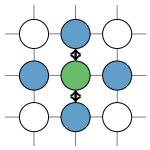
Organization. The rest of the paper is organized as follows. We start by introducing the process-to-node mapping problem in Section II and discuss related work in Section III. In Section IV, we look at the mapping problem for Cartesian graphs, for which we show that mapping problem for specific graph types is NP-hard. For that reason, we introduce three different, efficient heuristics to solve the process-to-node mapping problem for Cartesian graphs in Section V. We present the results of an extensive experimental evaluation of our novel algorithms in Section VI before we conclude in Section VII.

II. Notation and Problem Formulation

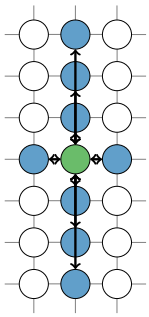
We are considering the traditional setup of HPC system architectures, where several compute nodes are interconnected via a high-speed network. The compute nodes usually comprise multiple processor-cores, often on two or more CPU sockets. In order to solve a computational problem on such systems, data needs to be exchanged among the distributed processes. We call communication between processes residing on different compute nodes *inter-node communication* and communication between processes residing on the same compute node *intra-node communication*. We follow the common assumption that intra-node communication (on a compute node) is (much) faster than inter-node communication with higher cumulated bandwidth. We assume homogeneous communication performance between the computation nodes, and also within the nodes [9], [10].



(a) nearest neighbor stencil



(b) component stencil



(c) nearest neighbor with hops stencil

Figure 2: Examples of the three 2-d stencils.

We denote by N the *number of (compute) nodes* allocated for the application to run. Let p be the total *number of processes* of an application and n_i with $i \in \{0, 1, \ldots, N-1\}$ the *number of processes per node* i, i.e., $\sum_{i=0}^{N-1} n_i = p$. If all the nodes have the same number of processes (homogeneous node sizes), we denote by n the number of processes on each node, i.e., $p = Nn$.

We assume that the processes are organized in a d-dimensional Cartesian grid with dimension sizes $\mathcal{D} = [d_0, \ldots, d_{d-1}]$, and thus, the *size of a grid* is the number of processes it comprises, $p = \prod_{i=0}^{d-1} d_i$. Each process with rank r, $0 \leq r < p$, is associated with a vector $\vec{r} = [r_0, \ldots, r_{d-1}]$, where $0 \leq r_i < d_i$ for $i \in \{0, 1, \ldots, d-1\}$, uniquely determining the position of the process in the grid. W.l.o.g., processes are assigned in row-major order to the grid.

Target Stencils: We now define three different stencils which will be used in the remainder of the article. A 2-d example of the considered stencils is depicted in Figure 2. To that end, we consider a *k-neighborhood* of a process to be a set of communication targets which can be described as a list of relative coordinates $\mathcal{S} = \{\vec{R_0}, \vec{R_1}, \ldots, \vec{R_{k-1}}\}$. Every $\vec{R_i} = [R_{i,0}, R_{i,1}, \ldots, R_{i,d-1}]$ with $0 \leq i < k$ describes the relative offset along the dimensions to the target process. Let $\mathbb{1}_i$ be a vector with the only non-zero component being one at index i. Then, we define the following stencils:

(a) *nearest neighbor stencil*: $\mathcal{S} = \{\mathbb{1}_i, -\mathbb{1}_i \mid 0 \leq i < d\}$,
(b) *component stencil*: $\mathcal{S} = \{\mathbb{1}_i, -\mathbb{1}_i \mid 0 \leq i < d-1\}$, and
(c) *nearest neighbor with hops*: $\mathcal{S} = \{\mathbb{1}_i, -\mathbb{1}_i \mid 0 \leq i < d\} \cup \{a\mathbb{1}_0, -a\mathbb{1}_0 \mid \forall a \in \{2, 3\}\}$.

Optimization Problem: By defining the k-neighborhood communication neighbors of each process in the Cartesian process grid with dimension sizes $\mathcal{D}$, we induce a Cartesian communication graph $C = (V, E)$ (Cartesian graph) where V is the vertex set representing the processes, i.e., $|V| = p$ and E is the set of communication edges between the processes. We assume unit edge-weight and sparse communication, i.e.,

the number of communication neighbors is much smaller than the total number of processes ($k \ll p$).

Let $\sigma : V \times V \to \{0, 1\}$ be a cost function that determines whether the communication between two processes v_i and v_j involves two different compute nodes, i.e., inter-node communication is required. Let $\mathcal{N}$ be the set of compute nodes and let $M : V \to \mathcal{N}$ be a function that maps a process $u \in V$ to exactly one compute node $\eta \in \mathcal{N}$. For all $u, v \in E$, let $\sigma(u, v) = 0$ if $M(u) = M(v)$ and $\sigma(u, v) = 1$ otherwise. The total cost (amount) of inter-node communication operations is defined as $J_{\text{sum}} := \sum_{(u,v)\in E} \sigma(u, v)$. We define the bottleneck node as the node with the largest number of outgoing communication edges, i.e., $N_b := \text{argmax}_{\eta\in\mathcal{N}}\{\sum_{(u,v)\in E} \sigma(u, v) \mid M(u) = \eta\}$. Let $\mathcal{B} := \{u \mid \forall u \in V : M(u) = N_b\}$ be the set of vertices assigned to the bottleneck node N_b. Then the cost of this bottleneck node is $J_{\text{max}} := \sum_{u\in\mathcal{B}} \sum_{(u,v)\in E} \sigma(u, v)$.

Our objective is to find a mapping function M of processes to nodes that minimizes J_{sum}. We use J_{max} to distinguish cases with similar values of J_{sum}, especially in the experimental evaluation. Note that the original allocation ($N \times n$) given by the scheduler needs to be respected, i.e., for each node N_i it must hold that $|\{u \in V \mid M(u) = N_i\}| = n_i$.

III. Related Work

There has been an immense amount of research on partitioning and process mapping – we refer to [11], [12], [13] for extensive material. The problem of process reordering for different topologies has been an active field of research since the beginning of `MPI` [14], [15], [16], [17], [18]. Many reordering algorithms take an arbitrary unstructured graph as input topology, making it difficult to perform efficient, scalable mappings on a structured grid, where communication is implicitly implied through the grid structure. In this paper we aim to exploit both stencil and grid structure, i.e., aim for specialized algorithms.

Gropp [9] pointed out that many `MPI` implementations have not implemented the `MPI_Cart_create` reordering method. As a response, he proposed an algorithm (`Nodecart`) for homogeneous node sizes n, based on the prime factorization of n. `Nodecart` decomposes the dimensions into a grid spanning the nodes and a grid describing the layout of the processes within a node. From this decomposition, every process can calculate its new coordinate $\vec{r}$ from which it can obtain its new rank. As a result, Gropp was able to show significant improvements in the time needed for a nearest neighbor message exchange in comparison to a blocked mapping of processes to nodes. `Nodecart` was specifically designed for the implied nearest neighbor stencil of Cartesian communicators in `MPI`.

Niethammer and Rabenseifner [10] used a different approach to assign processes to nodes. They point out that the `MPI_Dims_create` routine only considers the total number of processes p in an application from which it finds a grid decomposition where the dimension sizes are as close as possible. This algorithm can lead to bad domain decompositions in terms of inter-domain communication, if the underlying data mesh is not shaped cubically. Thus, they propose to solve the task of grid dimension creation and process mapping simultaneously. This is done by finding a factorization of the number nodes N, with the aim to minimize the weighted communication over the domain boundaries (the weights represent the expansion of the application mesh and a user defined communication cost factor). Their algorithm can be extended to handle hierarchical systems, where the weighted inter-domain communication is minimized at each level, but it requires symmetric hierarchies. With this approach, they can achieve significant performance gains for a nearest neighbor message exchange in comparison to the blocked assignment of processes to nodes.

Schulz et al. [19], [20] developed an algorithm (VieM, Vienna Mapping) for general process mapping with the objective of minimizing the total weighted communication. Their algorithm takes as input an unstructured communication graph and maps it onto a hierarchical hardware graph. This is done in a recursive manner with perfectly balanced graph partitioning techniques and randomized local search for improving found solutions. Even though the approach is costly in terms of runtime and memory, the communication cost in comparison to the state-of-the-art has been significantly reduced.

IV. NP-Hardness of Cartesian Mapping Problem

In general, graph embedding problems are NP-hard, as shown by [21]. However, the structure of the Cartesian graph mapping problem induced by a k-neighborhood pattern could make the problem easier to solve, in terms of NP-hardness or complexity of approximation algorithms. We propose the following formal definition of the Cartesian partitioning problem.

Definition IV.1. *Let $\mathcal{C} = (V, E)$ be a Cartesian graph with dimension sizes $\mathcal{D}$ and k-neighborhood $\mathcal{S}$, as defined in Section II and let $\mathcal{N}$ be a set of partition sizes (number of cores per compute node) $\mathcal{N} = n_0, \ldots, n_{N-1}$, s.t., $\sum_{i=0}^{N-1} n_i = |V|$. Let $M : V \to \mathcal{N}$ be a mapping function that assigns each vertex $v \in V$ of the Cartesian graph to a distinct partition $n \in \mathcal{N}$. The `GRID-PARTITION` problem answers the question whether there exists a mapping M such that $J_{sum} \leq Q$.*

Definition IV.2. *The `3-WAY-PARTITION` problem consists of dividing a multi-set of integers into three subsets, such that the sum of each subset is equal. Formally, given a multi-set $\mathcal{I}$ of integers, we ask whether $\mathcal{I}$ can be partitioned into 3 disjoint sub-sets $I_{1,2,3}$, where $\sum_{x\in I_1} x = \sum_{y\in I_2} y = \sum_{z\in I_3} z$ and $I_1 \cup I_2 \cup I_3 = \mathcal{I}$ holds.*

It is well-known that the `3-WAY-PARTITION` problem is NP-complete [22]. Now, we show that the `GRID-PARTITION` problem is already NP-hard for two dimensions and a simple, one-dimensional component stencil. To that end, we reduce `3-WAY-PARTITION` to `GRID-PARTITION` which leads to the following theorem.

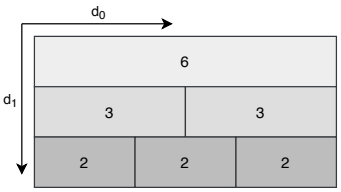


Figure 3: Example of 3-WAY-PARTITION to GRID--PARTITION transformation. Given $\mathcal{I}' = \{6,3,3,2,2,2\}$, create a grid with $\mathcal{D} = [6,2]$ for the Cartesian graph and find a mapping s.t. $J_{\text{sum}} \leq 2|\mathcal{I}'| - 6$.

Theorem IV.3. *The GRID-PARTITION problem is NP-hard, when restricted to two dimensions and a one-dimensional component stencil* $\mathcal{S} = \{-\mathbb{1}_1, \mathbb{1}_1\}$.

Proof. Given an arbitrary instance $\mathcal{I}'$ of 3-WAY--PARTITION, we construct an instance of GRID--PARTITION with a Cartesian graph, composed of $\mathcal{S}$ and $\mathcal{D}$, and $\mathcal{N}, Q$ as follows: $\mathcal{S} = \{-\mathbb{1}_1, \mathbb{1}_1\}$, $\mathcal{D} = [3, \frac{\sum_{x \in \mathcal{I}'} x}{3}]$, $\mathcal{N} = \{x \mid x \in \mathcal{I}'\}$, $Q = 2|\mathcal{I}'| - 6$.

An optimal mapping of a two-dimensional GRID--PARTITION problem with the component stencil always traverses the vertices in the grid along the communicating dimension given by the stencil, assigning them to a partition until it is full. Thus, for $|\mathcal{N}| \geq 3$, each partition has at most two outgoing communication edges (the first vertex assigned to the partition and the last), except partitions at the border of the grid which have one outgoing communication edge, i.e., we can say w.l.o.g that $Q = 2|\mathcal{N}| - 6$.

A *yes* instance of 3-WAY-PARTITION now corresponds to a *yes* instance of GRID-PARTITION, since we can assign every vertex in the first, second and third column to the partitions that correlate to the values in of I_1, I_2, I_3, respectively. □

An example for an instance with $\mathcal{I}' = \{6,3,3,2,2,2\}$ is shown in Figure 3. By encoding solutions with the first and last vertex of each partition, one can easily show that two-dimensional GRID-PARTITION with the component stencil is NP-complete (that is, is also in NP). With some adjustments, one can also show that the problem is NP-complete if we allow periodicity along the dimension of communication. The more interesting case is for which fixed stencils the problem remains NP-complete.

V. Rank Reordering Algorithms for k-Neighborhoods

In this section, we propose three algorithms for a k-neighborhood aware process reordering for Cartesian grids. The goal is to find reordering schemes that are a) fully distributed, that is, each process can compute its new rank independently of the other processes based on the input alone (grid and stencil), and b) efficient, that is, not polynomially dependent on p, preferably dependent only on the size of the (sparse, compared to p) stencil and the number of dimensions (we will tolerate polylog p).

A. Hyperplane Algorithm

The Hyperplane algorithm is a variation of recursive bisection. The main idea consists in recursively finding a split of a suitable dimension d_i of the Cartesian grid g into d_i', d_i'' s.t. $d_i = d_i' + d_i''$. This induces two grids g' and g'', where the ith dimension size of g' is d_i' and of g'' is d_i''. The split is chosen s.t. the sizes of the two new grids is a multiple of n, i.e., $n \mid |g'|$ and $n \mid |g''|$. Note that if we have heterogeneous nodes, one can use the mean, minimum or maximum of the node sizes as an input for the algorithm. This produces N grids each of size n which can be mapped to the nodes. The cuts should be chosen s.t. the minimal possible amount of communication between the grids is induced, since those correspond to inter-node communication. For that purpose, we calculate how parallel each vector $\vec{R} \in \mathcal{S}$ in the stencil is to a grid dimension j using the cosine.

$$\cos(\alpha_{\vec{R},\vec{e_j}}) = \frac{\vec{R}\vec{e_j}}{||\vec{R}||||\vec{e_j}||} \in [-1,1]. \tag{1}$$

Here, $\vec{e_j}$ is the unit vector along dimension j with $0 \leq j < d$, $\alpha_{\vec{R_i},\vec{e_j}}$ is the angle between the relative coordinate vector $\vec{R_i}$ and dimension j's unit vector $\vec{e_j}$. In order to have a monotonic increasing function, we square each of the values in Equation (1) and sum them over all relative coordinate vectors $\vec{R} \in \mathcal{S}$, giving us the following list

$$\left[\sum_{i=0}^{k-1} \cos^2(\alpha_{\vec{R_i},\vec{e_0}}),\ \ldots,\ \sum_{i=0}^{k-1} \cos^2(\alpha_{\vec{R_i},\vec{e_{d-1}}})\right]. \tag{2}$$

The dimension with the minimal value in Equation (2) is the most orthogonal to all $\vec{R} \in \mathcal{S}$, thus, we try to partition the grid alongside of it. Ties are broken by size, i.e., we want to partition along the bigger dimension. By sorting the dimensions according to their value in Equation (2), we define a preferred dimension order along which we try to perform the cuts.

The pseudo-code can be found in Algorithm 1. The input consists of the dimension sizes $\mathcal{D}$, the k-neighborhood $\mathcal{S}$, the number of processes per node n, the rank r of the calling process and it outputs the new position $\vec{r}_{\text{new}}$ of the calling rank on the Cartesian grid.

In each recursive step, we check if the grid is smaller than $2n$. We do this, for it is not needed to find an explicit cut for this grid size, rather, we can directly calculate the new coordinates with the preferred dimension order. This avoids bad splitting along very skewed grids, e.g., a nearest neighbor stencil on a two-dimensional grid with dimensions $[2,n]$ where n is large and odd. Instead of being forced to cut along the first dimension of size 2 to obtain two $[1,n]$ partitions, we obtain two partitions, each with 3 outgoing communication edges.

Otherwise, the algorithm finds the best possible split of the current grid into two new grids. For that purpose, it traverses the current dimensions $\mathcal{D}$ sorted in increasing order of the values defined in Equation (2) (sorting in each recursive step is necessary, because of changing dimension sizes) and tries to position the splitting hyperplane in the current dimension d_i.

Algorithm 1: `Hyperplane` Algorithm

Input: Grid with dimension sizes $\mathcal{D} = [d_0, \ldots, d_{d-1}]$, set of relative vectors $\mathcal{S}$ describing the k-neighborhood, number of processes per node n and rank r of calling process.

Result: New coordinate $\vec{r}_{\text{new}}$ of calling rank.

```
hyperplane(D, S, n, r, r_new)
1  if ∏_{d∈D} d ≤ 2n then
2  |  r_new ← new_coordinate(D, S, n, r)
3  |  return
4  else
5  |  i, d', d'' ← find_split(D, S, n)
6  |  LHS, RHS ← induced_grids(D, i, d', d'')
7  |  if r ∈ LHS then
8  |  |  hyperplane(LHS, S, n, r_new)
9  |  else
10 |  |  hyperplane(RHS, S, n, r_new)
11 |  end
12 end
```

The hyperplane is initially placed at the center $\frac{d_i}{2}$ of the candidate dimension d_i. If the initial split is not suitable, the position of the hyperplane is incremented/decremented, respectively, s.t. the position is as close as possible to the original grid's border in an effort to reduce $J_{\max}$. If it cannot find a suitable split along the candidate dimension, it will proceed to the next, until it finds a split. In Line 5 of Algorithm 1, the subroutine `find_split` returns the index i of the dimension to be split, and split sizes d' and d''. When a suitable split is found, two new grids g', g'' are created where the ith dimension size is replaced with d' and d'' for g' and g'', respectively. If the calling rank is located on the left-hand side of the split, it will call `Hyperplane` with g' (LHS) as input and else, it calls `Hyperplane` with g'' (RHS) as new input.

The number of recursions executed by the `Hyperplane` algorithm is logarithmic in the number of compute nodes N, although the two grids per recursion step can be very imbalanced in terms of size.

It follows that the running time of `Hyperplane` is bounded by $\mathcal{O}\left(\log N \sum_{i=0}^{d-1} d_i\right)$.

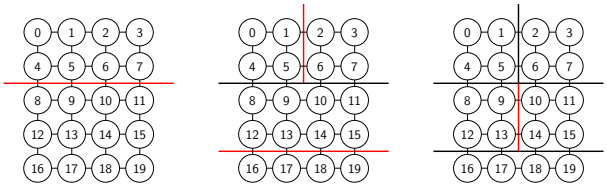
B. k-d Tree Algorithm

Similar to the `Hyperplane` algorithm, but inspired by the k-d tree data structure, the k-d `tree` algorithm works also by recursively splitting the grid along the dimensions. The main difference to the `Hyperplane` algorithm is that the recursion continues until there is only one vertex left in the Cartesian grid. The advantage of this approach is that the algorithm is no longer bound to the number of processes per node n. In fact, it is oblivious to it and only tries to find *dense* mappings, i.e., localize communicating vertices.

Instead of partitioning the grid dimensions in a round-robin manner, like in the k-d tree data-structure, we choose the largest dimension, weighted by the inverse amount of communication that is performed across it, so that we avoid splitting a dimension along which is communicated intensively. To be more precise, we define the amount of communication across a dimension j to be $f_j := |\{\vec{R} \mid \forall \vec{R} \in \mathcal{S} : R_j \neq 0\}|$, where R_j is the jth component of $\vec{R}$, then we want to find the dimension index $i := \text{argmax}_{0 \leq i < |\mathcal{D}|} \frac{d_i}{f_i}$ over the current grid dimensions sizes $\mathcal{D}$ and split equally along d_i. All ranks r with $r \leq \frac{1}{2} \prod_{j=0}^{d-1} d_j$ are assigned to the left-hand side, and the others to the right-hand side of the split.

The pseudo-code is given in Algorithm 2. If there is only one vertex left in the calling grid, we enter the base-case of the recursive function, in which the vector $\vec{r}_{\text{new}}$ already holds the coordinates of the calling process in the grid. Otherwise, we split the best suited dimension equally and recurse accordingly.

For the run-time analysis, it is not difficult to see that depth of the recursion tree is $\log_2 p$, where p is the total number of processes in the Cartesian grid. At each step of the recursion, we have to find the best dimension for the split. Using a priority queue, this can be achieved in $\mathcal{O}(\log d)$ steps.



(a) (b) (c)

Figure 4: `Hyperplane` algorithm example on 5×4 grid with a nearest neighbor stencil and $N = 5$, $n = 4$: (a) The first split is along the largest dimension; (b) Two new splits found on both grids; (c) Final split of the grid.

Algorithm 2: k-d `tree` Algorithm

Input: Dimension sizes $\mathcal{D}$, set of relative vectors describing the k-neighborhood $\mathcal{S}$ and rank r of calling process.

Result: New coordinate $\vec{r}_{\text{new}}$ of calling rank

```
kd_tree(D, S, r, r_new)
1  if ∏_{d∈D} d = 1 then
2  |  return
3  else
4  |  k ← find_split_index(D, S)
5  |  if r ≤ (∏_{d∈D} d)/2 then
6  |  |  d_k ← ⌊d/2⌋
7  |  |  kd_tree(D, S, r, r_new)
8  |  else
9  |  |  d_k ← ⌈d/2⌉
10 |  |  r_new[k] ← r_new + d_k
11 |  |  kd_tree(D, S, r, r_new)
12 |  end
13 end
```

Since the remaining computations are constant, the runtime for calculating the new reordering is $\mathcal{O}(\log p \log d)$.

C. Stencil Strips Algorithm

During early experimentation, we noticed that a consecutive assignment of processes to compute nodes for grids where the dimension sizes were close to the dth root of n (i.e., optimal side length) and the nearest neighbor stencil resulted in lower values of J_{sum} and $J_{\max}$ in comparison to recursive bisection. This observation was the motivation for this algorithm. The main idea is to partition the grid into strips, where strip lengths are chosen such that they are close to the scaled length of an optimal bounding rectangle of $\mathcal{S}$ (e.g., $\sqrt[d]{n}$ for the nearest neighbor stencil). Processes in coherent strips are assigned to partitions/nodes.

For that purpose, let R_i be the ith component of $\vec{R}$, then we define $n_{i,\max} := \max\{R_i \mid \forall \vec{R} \in \mathcal{S}\}$ and $n_{i,\min} := \min\{R_i \mid \forall \vec{R} \in \mathcal{S}\}$ to be the maximal and minimal value along dimension i in the k-neighborhood $\mathcal{S}$, respectively. The *extension* e_i of $\mathcal{S}$ along dimension i is $e_i := n_{i,\max} - n_{i,\min}$. We use the extensions to find a bounding n-dimensional rectangle of $\mathcal{S}$, with dimension sizes $\mathcal{E} = [e_0, \ldots, e_{d-1}]$. We define the volume V_b of the bounding n-dimensional rectangle to be

$$V_b := \prod_{i=0}^{d-1} \epsilon_i, \quad \epsilon_i = \begin{cases} 1 & \text{if } e_i = 0 \\ e_i & \text{else.} \end{cases}$$

With $d_b := |\{e_i \mid \forall e_i \in \mathcal{E} \wedge e_i \neq 0\}|$ being the number of non-zero expansions, we define the *distortion factor* α_i to a d_b-dimensional cube along dimension i as

$$\alpha_i := \frac{e_i}{\sqrt[d_b]{V_b}}.$$

As input for the algorithm serves the dimension sizes $\mathcal{D}$ of the grid, the k-neighborhood $\mathcal{S}$, the number of processes per node n and the rank r of the calling process. The pseudo-code is presented in Algorithm 3. The algorithm outputs the new position vector $\vec{r}_{\text{new}}$ of the calling rank. The algorithm itself works as follows: we calculate a list of distortion factors α for every dimension (Line 1). With the distortion factors, we can calculate a list s, containing the optimal strip length s_i along dimension i (Line 2), with consideration to already computed strip lengths, the distortion factors α_i and the node size n:

$$s_i := \sqrt[d-i]{\frac{\alpha_i n}{\prod_{j=0}^{i-1} s_j}}.$$

This is done for every dimension except the largest one, for we iteratively position the strips along the largest dimension. To be more precise, we assume that along the largest dimension the strip length is one. In every other dimension d_i, we fit $\lfloor \frac{d_i}{s_i} \rfloor$ strips (the last strips is of size $s_i + \frac{d_i}{s_i}$). Every process can locally compute the number of small and large strips along the dimensions, how many processes fit into each strip and, with the calling process' rank r the coordinates s_{coord} of the strip in which it is contained (Line 3-5), from which it can calculate its new position in the strip and thus, in the grid $\vec{r}_{\text{new}}$ (Line 9). In order to obtain cohesive partitions, we flip the strip assignment direction accordingly as depicted in Figure 5.

The strip widths, strip coordinates and the ranks position in the strip can each be calculated in $\mathcal{O}(d)$. The most expensive part of the algorithm is calculating the distortion factors, for which we need the maximal and minimal expansion along the dimensions. This amounts in $\mathcal{O}(kd)$ steps. Thus, the run-time of the algorithm is bounded by $\mathcal{O}(kd)$.

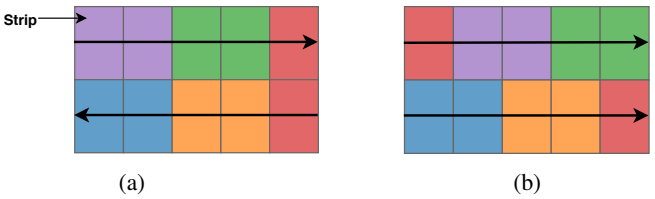
Algorithm 3: `Stencil Strips` Algorithm

Input: Dimension sizes $\mathcal{D}$, set of relative vectors $\mathcal{S}$ describing the k-neighborhood, number of processes per compute node n and rank r of calling process.

Result: New coordinate $\vec{r}_{\text{new}}$ of calling rank.

`stencil_strips` $(\mathcal{D}, \mathcal{S}, n, r, \vec{r}_{new})$

1 $\alpha \leftarrow$ `get_distortion_factors` $(\mathcal{S}, |\mathcal{D}|, n)$
2 $s \leftarrow$ `get_strip_lengths` $(|\mathcal{D}|, n, \alpha)$
3 $v \leftarrow \prod_{i \leftarrow 0}^{|\mathcal{D}|-1} s[i]$
4 **for** $i \leftarrow 0$ **to** $|\mathcal{D}| - 1$ **do**
5 $\quad v \leftarrow \frac{v}{s[i]}$
6 $\quad s_{\text{coord}}[i] \leftarrow \lfloor \frac{r}{v} \rfloor$
7 $\quad r \leftarrow r - (s_{\text{coord}}[i] - 1)v$
8 **end**
9 $\vec{r}_{\text{new}} \leftarrow$ `new_coordinate` $(\mathcal{D}, s, s_{coord}, r)$



(a) (b)

Figure 5: Assignment direction of the `Stencil Strips` algorithm. (a): Alternating strips assignment direction accordingly guarantees coherent partitions. (b): Imprudent assignment direction can lead to incoherent partitions.

VI. Experimental evaluation

We have implemented the presented algorithms for Cartesian rank reordering and the algorithm presented by Gropp [9], in accordance with the detailed pseudo-code of his paper. We aim to show the advantage of approaches that do not rely on factorization of the number of processes per node n. For that purpose, we compare the presented algorithms to a blocked assignment of ranks to nodes (henceforth referred as to blocked), `Nodecart`, and `VieM`, which are described in Section III. We start by describing the systems used to run the experiments in Section VI-A, implementation and benchmarking details are found in Section VI-B. In Section VI-C, we visualize the distribution of the reduction in inter-node communication achieved by the different algorithms over the blocked mapping for a wide set of instances. We proceed in Section VI-D, where we present the throughput gained by our algorithms

Table I: Machines used in the Experiments.

Name	Processor	MPI libraries	Compiler
VSC4	Intel Skylake Platinum 8174	Intel MPI	icc 19.0.5
SuperMUC-NG	Intel Skylake Platinum 8174	Intel MPI	icc 19.0.5
JUWELS	Intel Xeon Platinum 8168	Intel MPI	icc 19.0.3

for a neighbor all-to-all exchange and different message sizes on different machines. Finally, we compare the algorithmic running time in Section VI-E.

A. Machine Description

We perform the experiments in Section VI-D on the Vienna Scientific Cluster 4 (*VSC*4), *SuperMUC-NG* and *JUWELS*. *VSC*4 is composed of 790 nodes, where each node is equipped with two Intel Skylake Platinum 8174 processors running at 3.1 GHz, i.e., each node has 48 cores. The nodes are connected with a two-level fat-tree (blocking factor 2:1) via OmniPath with a capacity of 100 Gbit/s. *SuperMUC-NG* consists of 6336 compute nodes, also equipped with two Intel Skylake Platinum 8174 processors running at 3.1 GHz. The nodes are bundled into islands, with a pruned OmniPath connection between the islands (pruning factor 1:4) and nodes within an island are connected via a OmniPath fat-tree. *JUWELS* is composed of 2271 compute nodes, with two Intel Xeon Platinum 8168, running at 2.7 GHz. The nodes are connected with a two-level fat-tree InfiniBand network (pruning factor 2:1) with a capacity of 100 Gbit/s. A brief summary of the systems and libraries is given in Table I.

B. Experimental Setup

Listing 1: Interface used for a k-neighborhood aware `MPI` Cartesian communicator

```
int MPIX_Cart_stencil_comm(MPI_Comm oldcomm,
    const int ndims, const int dims[],
    const int periods[], const int reorder,
    const int stencil[], const int k,
    MPI_Comm *cartcomm);
```

All algorithms were implemented in C++ 11. The code was compiled with the Intel compiler and full optimization flags (-O3) and Intel MPI.

To remap for arbitrary stencils that cannot be expressed with the `MPI` Cartesian interfaces, we use an interface similar to `MPI_Cart_create` as shown in Listing 1. The array `stencil[]` consists of a flattened list of relative offsets along each dimension per neighbor. The number of neighbors is `k`, thus, the array `stencil[]` is of length $\texttt{k} \cdot \texttt{ndims}$.

As for the k-neighborhoods, we choose to test with the presented 2-dimensional stencils from Section II, i.e., nearest neighbor stencil, the nearest neighbor stencil with hops and the component stencil, as depicted in Figure 2.

All grids were created according to the `MPI_Dims_create` specifications, that is with the sizes of the dimensions being as close as possible to each other [8], [23].

For the stencil exchange, we instantiated a distributed graph communicator from the Cartesian communicator and the k-neighborhood in order to call the `MPI_Neighbor_alltoall` routine. Synchronization between the processes before each collective message exchange was done with an `MPI_Barrier` and we define the time needed for the operation, as the maximal time any process spent in the `MPI_Neighbor_alltoall` routine. As we assumed unit-weighted communication edges, every process sends and receives the same amount of data to its communication neighbors.

C. Inter-Node Communication Analysis

In this section, we investigate the reduction of J_{sum} and J_{max} of different algorithms over a blocked mapping. Note that the evaluation of the objective function is machine independent. We evaluate the performance of the algorithms on a varying input number of compute nodes, processes per compute node, and number of dimensions. To be more precise, the number of nodes is given by the set $\mathcal{N} = \{10, 13, 16, \ldots, 33\}$ and the number of processes per node by the set $\mathcal{P} = \{10, 13, 16, \ldots, 31\} \cup \{32\}$, while we restrict the set of dimensions to be $\mathcal{D} = \{2, 3\}$. The set of instances is therefore the Cartesian product $\mathcal{I} = \mathcal{N} \times \mathcal{P} \times \mathcal{D}$, with $|\mathcal{I}| = 144$. We define the reduction to be $\frac{C_X}{C_b}$, where C_X is the amount of inter-node communication induced by algorithm X and C_b the amount of inter-node communication induced by a blocked mapping. We compare both the total reduction of inter-node communication J_{sum}, as well as the reduction over the bottleneck node J_{max}, as defined in Section II. We evaluate the presented algorithms and compare ourselves to Gropp's algorithm (`Nodecart`) and `VieM`, both described in Section III. The configuration for `VieM` was set to the strongest setting, prioritizing quality in terms of reduction of inter-node communication instead of speed. As for the local-search neighborhood, we allowed swaps between any connected pair of vertices, i.e., we considered the largest search space. We set the `hierarchy_parameter_string` and the `distance_parameter_string` to the n:N and 0:1, respectively so that `VieM` optimizes our objective function.

The results for the three different k-neighborhood communication patterns can be seen in Figure 8. We plot the median improvement (orange) (due to its robustness to outliers) with a 95% confidence interval (indicated by the notches), calculated with a Gaussian-based asymptotic approximation. Since the confidence interval of the medians do not overlap, we can say with statistical evidence [24] that the median reduction improvement of `Hyperplane` and `Stencil Strips` is better than `Nodecart` in all communication patterns. Only for the nearest neighbor with hops pattern, there seems to be no statistical reduction between the k-d `tree` and `Nodecart`. For the nearest neighbor and component stencil, the reduction distribution between the `Stencil Strips` and `VieM` looks very similar and there is no statistical difference in the median reduction, indicating that the `Stencil Strips` algorithm, which is not bound to factorization in any way, can greatly exploit the structure of the grid partitioning problem.

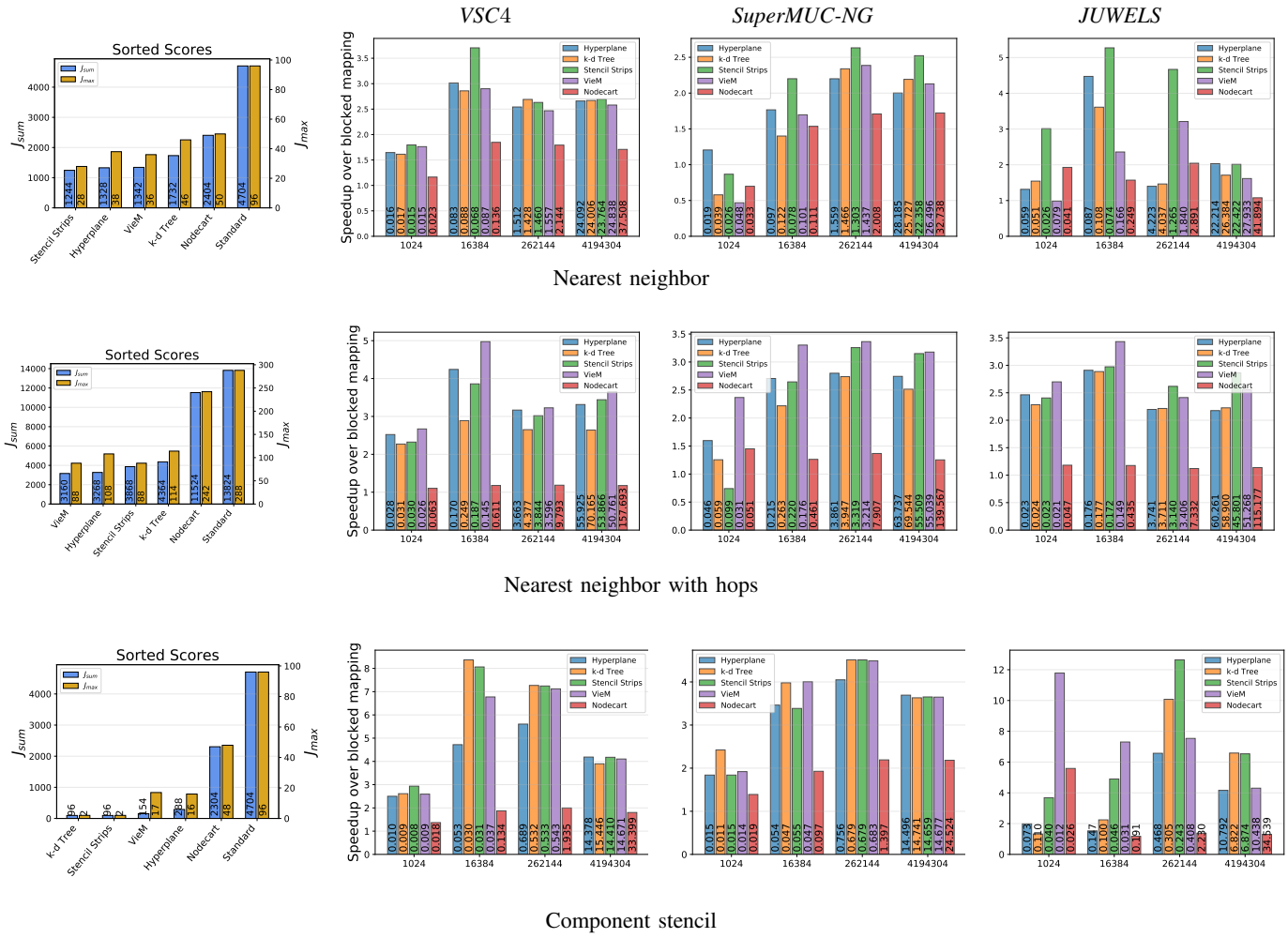


Figure 6: Left column: scores of the algorithms (smaller is better). Right three columns: Speedup over blocked mapping (higher is better) with $N = 50$ number of nodes and $p = 48$ processes per node and grid sizes 50×48 for the nearest neighbor stencil, the nearest neighbor stencil with hops and the component stencil on the *VSC*4, *SuperMUC-NG*, and *JUWELS*. Absolute times are written in the corresponding bar.

D. Throughput Analysis

We continue by examining the influence of rank reordering on the time needed for an `MPI_Neighbor_alltoall` exchange and different message sizes to be sent to each communication partner. The experiment was conducted on *VSC*4, *SuperMUC-NG*, and *JUWELS* described in Section VI-A. In order to see how the reordering affects the communication performance of larger instances, we perform the experiments with 50 and 100 nodes and for each used the full number of processes per node 48. Each message was sent 200 times to obtain large samples. From each run, we capture the maximum time needed over all processes. Before each exchange, we synchronize the processes with `MPI_Barrier`. In Figure 6 and Figure 7, we plot the mean speedup over the blocked assignments, after removing outliers beyond 1.5 inter-quartile range from the third and first quartile, respectively. Along with the speedup, we plot the scores of the algorithms sorted in increasing order of J_{sum} and J_{max}. The message size is the number of bytes send per neighbor.

For the nearest neighbor stencil and $N = 50$, Figure 6, `Hyperplane` and `Stencil Strips` are up to three and four times faster than the blocked assignment on the *VSC*4, up to two and almost up to three times faster on *SuperMUC-NG* and more than five times faster on *JUWELS*. All of the presented algorithms are able to outperform `Nodecart` in terms of the mapping metric J_{sum}, J_{max} and communication performance on *VSC*4 and mostly on *SuperMUC-NG* and *JUWELS*. As for `VieM`, `Stencil Strips` outperforms it consistently on all machines, whereas `Hyperplane` and k-d `tree` attain similar or better performance. Surprisingly, the communication performance of `VieM` on *JUWELS* is worse than predicted by J_{sum}, J_{max}. In the case of $N = 100$, Figure 7, `Hyperplane` and `Stencil Strips` obtain speedups of up to three over the blocked assignment on *VSC*4, up to 2.5 on *SuperMUC-*

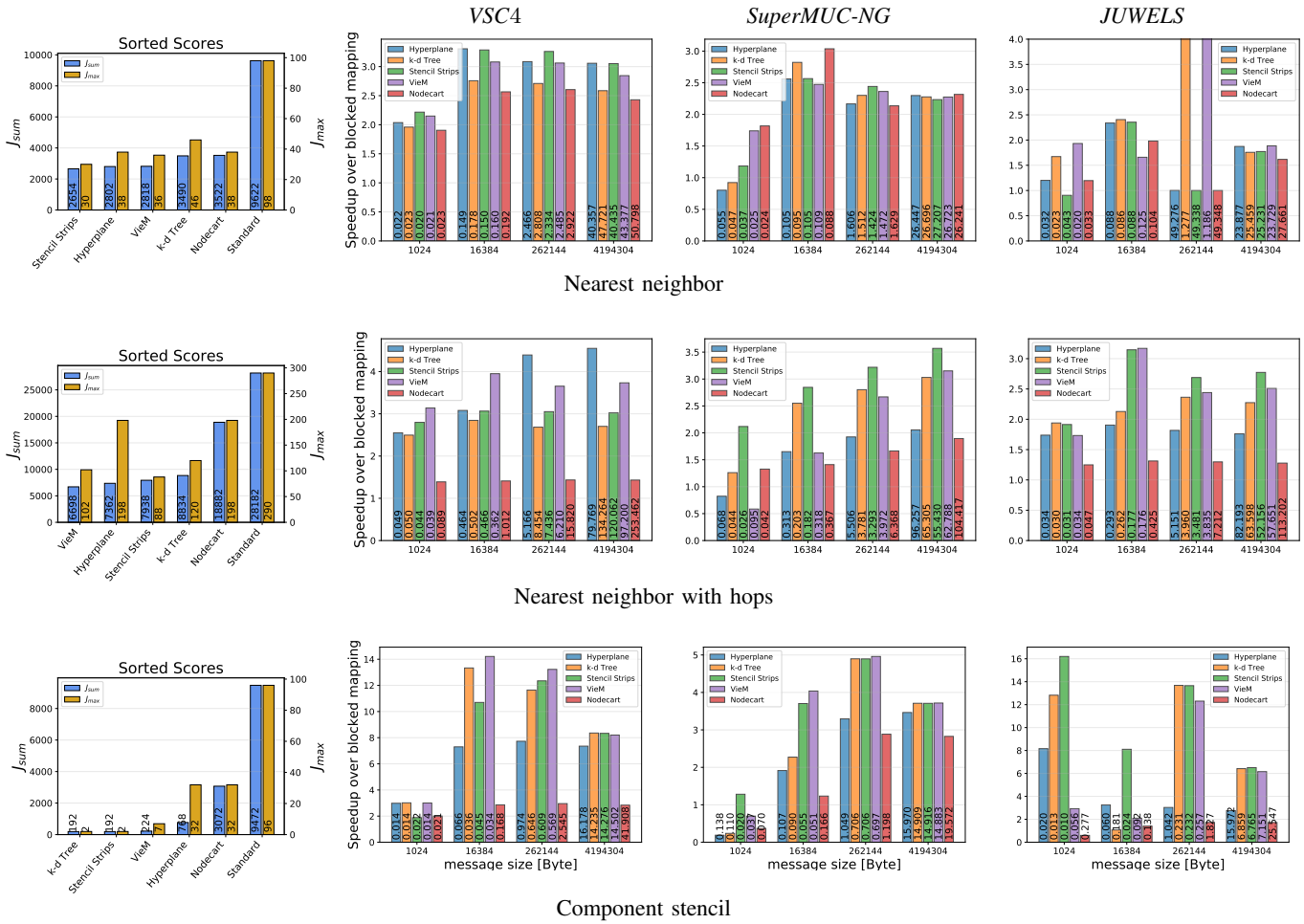
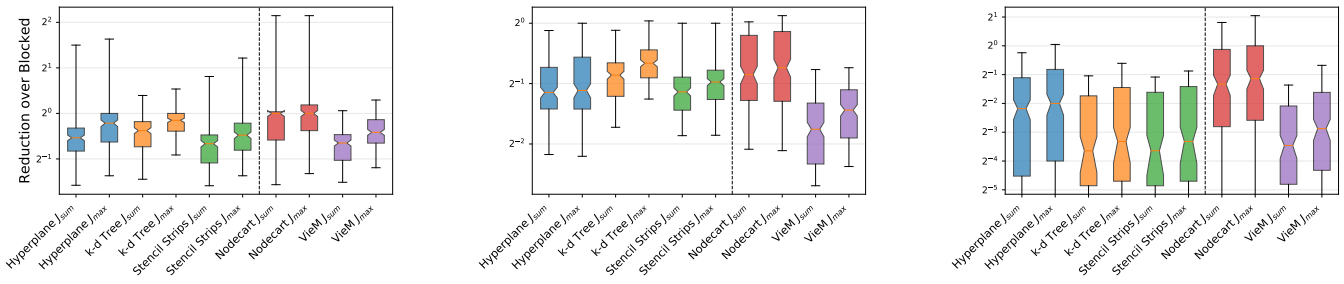


Figure 7: Left column: scores of the algorithms (smaller is better). Right three columns: Speedup over blocked mapping (higher is better) with $N = 100$ number of nodes and $p = 48$ processes per node and grid sizes 75×64 for the nearest neighbor stencil, the nearest neighbor stencil with hops and the component stencil on the *VSC*4, *SuperMUC-NG*, and *JUWELS*. Absolute times are written in the corresponding bar.

NG and up to 2 on *JUWELS*. For message size 262 144 the performance of the blocked assignment, `Hyperplane`, `Stencil Strips`, and `Nodecart` performed very badly, resulting in speedups of more than forty, which is why we cut the axis at 4.

In the case of the nearest neighbor with hops stencil for $N = 50$, Figure 6, `Hyperplane`, `Stencil Strips` obtained speedups of up to four on the *VSC*4, up to 3.5 on *SuperMUC-NG* and up to three on *JUWELS*. `VieM` outperforms the presented algorithms and both *VSC*4 and *SuperMUC-NG*, but `Stencil Strips` obtained similar to better performance for large message sizes on *JUWELS*. In comparison to `Nodecart`, the presented algorithms seemed to be up to two and three times faster. For $N = 100$, Figure 7, the presented algorithms are up to a factor 4.5 faster than the blocked assignment on *VSC*4, up to 3.5 faster on *SuperMUC-NG* and more than a factor 2 faster on *JUWELS*. On the *VSC*4, our algorithms are more than a factor two faster than `Nodecart`, whereas on *SuperMUC-NG* and *JUWELS* we outperform `Nodecart` for most message sizes by a factor of more than 1.3.

As for the synthetic component stencil and $N = 50$, only k-d `tree` and `Stencil Strips` managed to find an optimal mapping, where each compute node has two outgoing communication edges, as described in Section IV. This reordering results in speedups of up to eight over the blocked assignment on *VSC*4, up to four on *SuperMUC-NG* and up twelve on *JUWELS*. Interestingly, `VieM` seems to have similar performance on *VSC*4 and *SuperMUC-NG*, and outperforms our algorithms for small message sizes on *JUWELS*. In the case of $N = 100$, again only k-d `tree` and `Stencil Strips` found optimal mappings, leading to speedups of up to fourteen on *VSC*4, five on *SuperMUC-NG*, and sixteen on *JUWELS*. Surprisingly, `VieM` seems to match and outperform the performance of our algorithms, even though it has a factor 1.6 more J_{sum} and a factor 8.5 greater



(a) (b) (c)

Figure 8: Reduction for nearest neighbor stencil (a), for the nearest neighbor stencil with additional two hops along the first dimension (b) and for the artificial component stencil (c). Lower is better. For each algorithm `sum` defines the total reduction, whereas `max` defines the reduction of the bottleneck node. `Hyperplane` depicted in blue, k-d `tree` in yellow, `Stencil Strips` in green, `Nodecart` in red, and `VieM` as comparison in gray.

$J_{\max}$.

E. Instantiation Time

Since the theoretical complexity of the three presented algorithms, `Nodecart` and `VieM` all dependent on different parameters, we benchmark the algorithmic runtime needed to calculate the new ranks only on the largest nearest neighbor stencil instance, described in Section VI-D ($N = 100$) on *VSC4*. The algorithmic runtime does not include the instantiation time of the reordered Cartesian communicator, rather, it only includes the necessary operations to calculate the new ranks. For the presented algorithms, this includes creating a sorted communicator, with precise knowledge over which ranks are assigned to which nodes. This custom communicator consists of a *node communicator* (grouped by processes having shared memory) and a *leader communicator*, which consists of one process per compute node. In this benchmark, the k-d `tree` algorithm was implemented with a linear search, for finding the dimension along which to split, resulting in a theoretical run-time of $\mathcal{O}(d \log p)$. Each algorithm was instantiated 200 times for large sample sizes, and we measured the longest time needed over all processes. Before each instantiation, the processes were synchronized using `MPI_Barrier`. In Figure 9, we plot the mean instantiation time, after outlier removal (beyond 1.5 inter-quartile range from the first and third quartile) with a 95% confidence interval. We did not include `VieM`, since it took on average over 7.95 s (a factor of more than 400) and it would distort the scale of the plot.

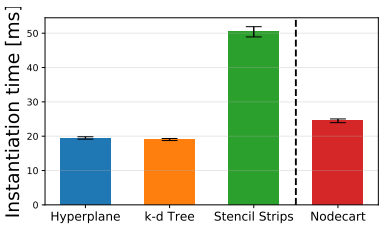


Figure 9: Instantiation time in ms on a 100×48 nearest neighbor stencil instance, run on *VSC4* (lower is better). Each algorithm was instantiated 200 times and after outlier removal, we plot the mean with a 95% confidence interval. `VieM` was omitted since it took two orders of magnitude longer and would distort the scale.

We can see, that for this instance the `Hyperplane` and k-d `tree` algorithm are the two fastest, and statistically equivalent. `Nodecart` seems to need 28% longer, whereas the convoluted process of calculating the strips and strip positions results in `Stencil Strips` being the slowest algorithm, more than a factor 2 slower than `Hyperplane` and k-d `tree`.

VII. Conclusion

We introduced three new efficient algorithms for process to compute node assignment on Cartesian grids and stencils communication patterns. By thoroughly exploiting the inherently present structure of the problem, we arrive at algorithms that outperform the state-of-the-art in terms of running time and communication performance. We implemented the algorithms for `MPI_Cart_create` as reordering functions and performed extensive benchmarks. An intensive experimental evaluation shows that our algorithms are up to two orders of magnitude faster in running time than a (sequential) high-quality general graph mapping tool `VieM`, while obtaining similar results in communication performance. Furthermore, our algorithms are three times faster in an `MPI_Neighbor_alltoall` exchange than a state-of-the-art Cartesian grid mapping algorithm (`Nodecart`) by achieving a significantly better mapping quality. Considering the good results, we plan to release the algorithms and integrate them into publicly available `MPI` implementations.

Acknowledgments

This work was partially supported by the Austrian Science Fund (FWF): project P 31763-N31. The computational results presented have been achieved in part using the Vienna Scientific Cluster (VSC). We acknowledge PRACE for awarding us access to JUWELS at GCS@FZJ, Germany and SuperMUC-NG at GCS@LRZ, Germany.

References

[1] R. M. Haralick and L. G. Shapiro, *Computer and robot vision*. Addison-Wesley, 1992, vol. 1.

[2] A. Taflove and S. C. Hagness, *Computational electrodynamics: the finite-difference time-domain method*. Artech House, 2005.

[3] S. D. Ustyugov, M. V. Popov, A. G. Kritsuk, and M. L. Norman, "Piecewise parabolic method on a local stencil for magnetized supersonic turbulence simulation," *J. Comput. Phys.*, vol. 228, no. 20, pp. 7614–7633, 2009.

[4] A. Sawdey, M. O'Keefe, R. Bleck, and R. W. Numrich, "The design, implementation, and performance of a parallel ocean circulation model," in *Proceedings of 6th ECMWF Workshop on the Use of Parallel Processors in Meteorology: Coming of Age*, 1995, pp. 523–550.

[5] L. Renganarayanan, M. Harthikote-Matha, R. Dewri, and S. V. Rajopadhye, "Towards optimal multi-level tiling for stencil computations," in *Proceedings of the 21th International Parallel and Distributed Processing Symposium (IPDPS)*, 2007, pp. 1–10.

[6] M. Christen, O. Schenk, and Y. Cui, "Patus for convenient high-performance stencils: Evaluation in earthquake simulations," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, 2012, p. 11.

[7] J. Fietz, M. J. Krause, C. Schulz, P. Sanders, and V. Heuveline, "Optimized hybrid parallel lattice boltzmann fluid flow simulations on complex geometries," in *Proceedings of the 18th European Conference on Parallel Processing (Euro-Par)*, ser. LNCS. Springer, 2012, pp. 818–829.

[8] MPI Forum, *MPI: A Message-Passing Interface Standard. Version 3.1*, June 4th 2015, http://www.mpi-forum.org.

[9] W. D. Gropp, "Using node and socket information to implement MPI cartesian topologies," *Parallel Computing*, vol. 85, pp. 98–108, 2019.

[10] 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, pp. 6:1–6:8.

[11] C. Bichot and P. Siarry, Eds., *Graph Partitioning*. Wiley-ISTE, 2011.

[12] A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz, *Recent Advances in Graph Partitioning*, ser. LNCS. Springer International Publishing, 2016, vol. 9220, pp. 117–158. [Online]. Available: https://doi.org/10.1007/978-3-319-49487-6_4

[13] C. Schulz and D. Strash, "Graph partitioning: Formulations and applications to big data," in *Encyclopedia of Big Data Technologies*, Z. A. Sakr S., Ed. Springer, Cham, 2019. [Online]. Available: https://doi.org/10.1007/978-3-319-63962-8_312-2

[14] B. Brandfass, T. Alrutz, and T. Gerhold, "Rank reordering for mpi communication optimization," *Computers & Fluids*, vol. 80, pp. 372–380, 2013.

[15] J. L. Träff, "Implementing the MPI process topology mechanism," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2002.

[16] G. Mercier and E. Jeannot, "Improving MPI applications performance on multicore clusters with rank reordering," in *Proceedings of the 18th European MPI Users' Group Meeting (EuroMPI)*, ser. LNCS, vol. 6960. Springer, Berlin, Heidelberg, 2011. [Online]. Available: https://doi.org/10.1007/978-3-642-24449-0_7

[17] T. Hatazaki, "Rank reordering strategy for MPI topology creation functions," in *Proceedings of the 5th European PVM/MPI Users' Group Meeting (EuroMPI/PVM)*, ser. LNCS, vol. 1497. Springer, Berlin, Heidelberg, 1998.

[18] H. Yu, I.-H. Chung, and J. E. Moreira, "Topology mapping for Blue Gene/L supercomputer," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. ACM, 2006, p. 116.

[19] C. Schulz and J. L. Träff, "Better process mapping and sparse quadratic assignment," in *Proceedings of the 16th International Symposium on Experimental Algorithms (SEA)*, 2017, pp. 4:1–4:15.

[20] C. Schulz, J. L. Träff, and K. von Kirchbach, "Better process mapping and sparse quadratic assignment," *CoRR*, vol. abs/1702.04164, 2017. [Online]. Available: http://arxiv.org/abs/1702.04164

[21] S. H. Bokhari, "On the mapping problem," *IEEE Transactions on Computers*, vol. 30, pp. 207–214, 1981.

[22] R. E. Korf, "Multi-way number partitioning," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI)*. Morgan Kaufmann Publishers Inc., 2009, pp. 538—-543.

[23] J. L. Träff and F. D. Lübbe, "Specification guideline violations by `MPI_Dims_create`," in *Proceedings of the 22nd European MPI Users' Group Meeting (EuroMPI)*, 2015, pp. 19:1–19:2.

[24] A. Murphy and R. Katz, *Probability, Statistics, And Decision Making In The Atmospheric Sciences*. CRC Press, 2019.