Better Process Mapping and Sparse Quadratic Assignment[*]

Christian Schulz[1] and Jesper Larsson Träff[2]

1   Karlsruhe Institute of Technology, Karlsruhe, Germany; and
    University of Vienna, Vienna, Austria
    `christian.schulz@kit.edu, christian.schulz@univie.ac.at`
2   TU Wien, Vienna, Austria
    `traff@par.tuwien.ac.at`

Abstract

Communication and topology aware process mapping is a powerful approach to reduce communication time in parallel applications with known communication patterns on large, distributed memory systems. We address the problem as a quadratic assignment problem (QAP), and present algorithms to construct initial mappings of processes to processors as well as fast local search algorithms to further improve the mappings. By exploiting assumptions that typically hold for applications and modern supercomputer systems such as sparse communication patterns and hierarchically organized communication systems, we arrive at significantly more powerful algorithms for these special QAPs. Our multilevel construction algorithms employ recently developed, perfectly balanced graph partitioning techniques and excessively exploit the given communication system hierarchy. We present improvements to a local search algorithm of Brandfass et al. (2013), and decrease the running time by reducing the time needed to perform swaps in the assignment as well as by carefully constraining local search neighborhoods. Experiments indicate that our algorithms not only dramatically speed up local search, but due to the multilevel approach also find much better solutions in practice.

1   Introduction

Communication performance between processes in high-performance systems depends on many factors. For example, communication is typically faster if communicating processes are located on the same processor node compared to the cases where processes reside on different nodes. This becomes even more pronounced for large supercomputer systems where processors are hierarchically organized into, e. g., islands, racks, nodes, processors, cores with corresponding communication links of similar quality. Given the communication pattern between processes and a hardware topology description that reflects the quality of the communication links, one hence seeks to find a good mapping of processes onto processors such that pairs of processes exchanging large amounts of data are located closely.

Such a mapping can be computed by solving a corresponding quadratic assignment problem (QAP) which is a hard optimization problem. Sahni and Gonzalez [26] have shown

---

QAP to be strongly NP-hard and, unless P=NP, admitting no polynomial-time constant factor approximation algorithm. In addition, there are no algorithms that can solve meaningful instances with $n > 20$ to optimality in a reasonable amount of time [9]. Hence, heuristic algorithms are necessary in order to solve large scale instances. Multiple heuristics have been proposed to tackle real world instances [7, 17, 23]. We present more details in Section 3.

In this work, we make two important assumptions that are typically valid for modern supercomputers and the applications that run on those. First, communication patterns are almost always sparse since not all processes have to communicate with each other. This is especially true for large scale scientific simulations in which the underlying models of computation and communication are already sparse, see, e. g., [10, 14, 28]. To efficiently parallelize the simulation one normally employs graph partitioning techniques which then in turn yield a sparse communication pattern between the processes. Second, we assume that the hardware communication topology under consideration is hierarchical with communication links on the same level in the hierarchy having the same communication speed. This is typically observed in current high-performance systems, e. g., SuperMUC[1].

Using these assumptions, we derive algorithms that are able to create high quality mappings, as well as faster local search algorithms for improving assignments. Overall, our algorithms are able to compute better solutions than other recent heuristics for the problem. Improving the (practical) complexity of such algorithms is highly important, since the number of cores available in supercomputers is still increasing dramatically. The rest of this paper is organized as follows. In Section 2, we introduce basic concepts and describe relevant related work, such as the algorithm of Brandfass et al. [7], in more detail. We present our main contributions in Section 3. We implemented the techniques presented here in the graph partitioning framework KaHIP [27] (Karlsruhe High Quality Graph Partitioning). A summary of extensive experiments to evaluate algorithm performance is presented in Section 4. Experiments indicate that our algorithm not only drastically speeds up local search, but due to the multilevel approach that employs recently developed high quality partitioning techniques also finds better solutions in practice.

2   Preliminaries

The total communication requirement between the set of processes in (some section of) an application can be modeled by a weighted communication graph. The underlying hardware topology can likewise be modeled by a weighted graph, but since the graph is complete (any physical processor can communicate with any other physical processor through the underlying networks), we represent it by a topology cost matrix which can for instance reflect the costs of routing along shortest (cheapest) paths between processes. Our abstract problem is to embed the communication graph onto the topology graph under optimization criteria that we explain below. We assume that the number of nodes in host and topology graphs are the same. Unless otherwise mentioned, a processing element (PE) typically represents a core of a machine.

Basic Concepts.   In the following, we consider an undirected graph $G = (V = \{0, \ldots, n-1\}, E)$ with edge weights $\omega : E \to \mathbb{R}_{>0}$, node weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We extend c and ω to sets, i. e., $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. We let $\Gamma(v) := \{u : \{v, u\} \in E\}$ denote the neighbors of a node v. A graph $S = (V', E')$ is said

---

to be a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. We call S an *induced* subgraph when $E' = E \cap (V' \times V')$.

Throughout the paper, $\mathcal{C} \in \mathbb{R}^{n \times n}$ denotes the communication matrix and $\mathcal{D} \in \mathbb{R}^{n \times n}$ the hardware topology matrix or distance matrix. More precisely, $\mathcal{C}_{i,j}$ describes the amount of communication that has to be done between process i and j, and $\mathcal{D}_{i,j}$ represents the weighted distance between PE i and PE j. That is, the cost for communicating the amount $\mathcal{C}_{i,j}$ between processors i and j is $\mathcal{C}_{i,j}\mathcal{D}_{i,j}$. We follow Brandfass et al. [7] and others, and model the embedding problem as a quadratic assignment problem (QAP): Find a one-to-one mapping Π of processes to PEs which minimizes the overall communication cost. More precisely, we want to minimize $J(\mathcal{C}, \mathcal{D}, \Pi) := \sum_{i,j} \mathcal{C}_{\Pi(i),\Pi(j)}\mathcal{D}_{i,j}$ where the sum is over all PE pairs and $k = \Pi(i)$ means that process k is assigned to PE i. Note that searching for the inverse permutation instead, i.e., assigning process i to PE $\Pi^{-1}(i)$, results in the same assignment problem as Π is a one-to-one mapping. Throughout this work, we assume that $\mathcal{C}$ and $\mathcal{D}$ are symmetric – otherwise one can create equivalent QAP problems with symmetric inputs [7]. In this paper, we focus on *sparse* communication patterns, and therefore do not want to store the complete communication matrix but instead represent it more efficiently as a graph. On the other hand, typical system topologies feature a hierarchy that we can exploit. Hierarchy information, and in general $\mathcal{D}$, is given implicitly and can be queried, and therefore does not have to be stored explicitly.

Graph partitioning is a key component in our algorithms to find initial solutions. The *graph partitioning problem* looks for *blocks* of nodes $V_1,\ldots,V_k$ that partition V, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *balancing constraint* demands that $\forall i \in 1..k : c(V_i) \leq L_{\max} := (1 + \epsilon)\lceil c(V)/k \rceil$ for some parameter ϵ. In the *perfectly balanced case* the imbalance parameter ϵ is set to zero, i.e., no deviation from the average is allowed. One commonly used objective is to minimize the total *cut* $\sum_{i<j} w(E_{ij})$ where $E_{ij} := \{\{u,v\} \in E : u \in V_i, v \in V_j\}$.

Related Work. There has been a *huge* amount of research on GP, and we refer the reader to [4, 8] for extensive material and references. All general-purpose methods that work well on large real-world graphs are based on the multilevel principle. The basic idea can be traced back to multigrid solvers for systems of linear equations [31]. Well-known MGP software packages include Jostle [34], Metis [19], and Scotch [25]. Jostle contains algorithms to compute processor assignments in scientific simulations. Jostle integrates local search into a multi-level method to partition the model of computation and communication. To do so, they solve the problem on the coarsest level and afterwards perform refinement that takes the user supplied network communication model into account. Scotch performs dual recursive bipartitioning to perform this task. There is likewise a large literature on process mapping [6, 24], often in the context of scientific applications using MPI [22] (Message-Passing Interface). Bokhari shows that the mapping problem is equivalent to the graph isomorphism problem [5]. Hatazaki [16] was among the first authors to propose graph partitioning to solve the process mapping problem for unweighted process topologies in the specific context of MPI. Träff [32] used a similar approach, and gave one of the first non-trivial implementations for the NEC vector supercomputers. Mercier and Clet-Ortega and later Jeannot [20, 21] simplify the mapping problem by only considering the topology inside the compute nodes themselves and ignoring the topology of the network. Multiple placement policies are investigated to enhance overall system performance. Yu et al. [35] discuss and implement embedding heuristics for the BlueGene $3d$ torus systems. Hoefler and Snir [18] optimize instead the congestion of the mapping, show that this problem is NP-complete, and give a corresponding heuristic with an experimental evaluation based on application data from the Florida Sparse Matrix Collection.

Routing aware mapping heuristics taking the hierarchy of specific hardware topologies into account were discussed in [1]. A resource-aware graph partitioning framework has been proposed by Chan et al. [11]. Vogelstein et al. [33] concentrate on solving general quadratic assignment and graph matching problems. They propose a gradient based heuristic that involves solving assignment problems and give experimental evidence for better solution quality and speed compared to certain other heuristics. The worst-case complexity of their approach is $O(n^3)$ steps.

Detailed Related Work.   We now discuss related work by Müller-Merbach [23], Heider [17] and Brandfass et al. [7] as well as Glantz et al. [15] in greater detail since our work either makes use of the tools proposed by those authors or because we compare against their results. Müller-Merbach [23] proposes a greedy construction method to obtain an initial permutation for the QAP. The method roughly works as follows: Initially compute the total communication volume for each processor and also the total distance for each core. Note that this corresponds to the weighted degrees of the vertices in the communication and distance models, respectively. Afterwards, the process with the largest communication volume is assigned to the core with the smallest total distance. To build a complete assignment, the algorithm proceeds by looking at unassigned vertices and cores. For each of the unassigned processes the communication load to already assigned vertices is computed. For each core, the total distance to already assigned cores is computed. The process with the largest communication sum is assigned to the core with the smallest distance sum. Glantz et al. [15] note that the algorithm does not link the choices for the vertices and cores and propose a modification of this algorithm called *GreedyAllC* (the best algorithm in [15]). GreedyAllC links the mapping choices by scaling the distance with the amount of communication to be done. The algorithm has the same asymptotic complexity and memory requirements as the algorithm by Müller-Merbach. We also compare our proposed methods against GreedyAllC in Section 4.

Heider [17] proposes a method to improve an already given permutation/mapping. The method repeatedly tries to perform swaps in the assignment. To do so, the author defines a pair-exchange neighborhood $N(\Pi)$ that contains all permutations that can be reached by swapping two elements in Π. Here, swapping two elements means that $\Pi(i)$ will be assigned to processor j and $\Pi(j)$ will be assigned to processor i after the swap is done. The algorithm then looks at the neighborhood in a cyclic manner. More precisely, in each step the current pair (i, j) is updated to $(i, j + 1)$ if $j < n$, to $(i + 1, i + 2)$ if $j = n$ and $i < n - 1$, and lastly to $(1, 2)$ if $j = n$ and $i = n - 1$. A swap is performed if it yields positive gain, i. e., the swap reduces the objective. The overall runtime of the algorithm is $O(n^3)$. We denote the search space with N^2. To reduce the runtime, Brandfass et al. [7] introduce a couple of modifications. First of all, only symmetric inputs are considered. If the input is not symmetric, the input is substituted by a symmetric one such that the output of the algorithm remains the same. Second, pairs (i, j) for which the objective cannot change, are not considered. For example, if two processes reside on the same compute node, swapping them will not change the objective. Lastly, the authors partition the neighborhood search space into s consecutive index blocks and only perform swaps inside those blocks. This reduces the number of possible pairs from $O(n^2)$ to $O(ns)$ overall pairs. We denote the search space with $\mathcal{N}_p$ (*pruned neighborhood*). In addition, instead of starting from the identity permutation, the authors use the method of Müller-Merbach [23] to compute an initial solution. This improves runtime of the local search approach as well as the objective of the solution.

3   Rank Reordering Algorithms

We now present our main contributions and techniques. This includes algorithms to compute initial solutions, speeding up the local search algorithms for sparse communication patterns and defining new search spaces for the local search algorithm. Throughout this section, we assume that the input communication matrix is already given as a graph $G_\mathcal{C}$, i.e., no conversion of the matrix into a graph is necessary. More precisely, the graph representation is defined as $G_\mathcal{C} := (\{1, \ldots, n\}, E[\mathcal{C}])$ where $E[\mathcal{C}] := \{(u, v) \mid \mathcal{C}_{u,v} \neq 0\}$. In other words, $E[\mathcal{C}]$ is the edge set of the processes that need to communicate with each other. Note that the set contains forward and backward edges, and that the weights of the edges in the graph correspond to the entries in the matrix $\mathcal{C}$.

3.1   Initial Solutions

We propose two strategies exploiting the hierarchy. Intuitively, we want to identify subgraphs in the communication graph of processes that have to communicate much with each other and then place such processes closely, i.e., on the same node, same rack and so forth. In the following, we assume a homogeneous hierarchy of the supercomputer, but our algorithms can be extended to heterogeneous hierarchies in a straightforward way. Let $\mathcal{S} = a_1, a_2, ..., a_k$ be a sequence describing the hierarchy of the supercomputer (with $n = \Pi_i a_i$). The sequence should be interpreted as each processor having a_1 cores, each node a_2 processors, each rack a_3 nodes, ... We propose two algorithms to compute initial mappings, a top down and a bottom up approach. The first one, *top down*, splits the communication graph recursively and the second one, builds a hierarchy *bottom up*.

The *top down approach* starts by computing a *perfectly balanced* partition of $G_\mathcal{C}$ into a_k blocks each having n/a_k vertices (processes). The partitioning task is done using the techniques provided by Sanders and Schulz [27] which provide high quality partitions and guarantee that each block of the output partition has the specified amount of vertices. In principle, the nodes of each block will be assigned completely to one of the a_k system entities. Each of the system entities provides precisely n/a_k PEs. We then proceed recursively and partition each subgraph induced by a block into a_{k-1} blocks and so forth. The recursion stops as soon as the subgraphs have only a_1 vertices left. In the base case, we assign processes to permutation ranks.

The *bottom up approach* proceeds in the opposite order of the hierarchy. That means the communication graph $G_\mathcal{C}$ is split first into $k = n/a_1$ blocks with precisely a_1 vertices each. Again, this is done using the perfectly balanced partitioning techniques mentioned above. Each block will later on be assigned to a unique system entity that is able to host a_1 processes, i.e., a node having a_1 cores. Then each of the blocks is contracted and we partition the contracted graph and so forth. In this case, if replacing edges of the form $\{u, w\}, \{v, w\}$ would generate two parallel edges $\{x, w\}$, we insert a single edge with $\mathcal{C}'_{x,w} = \mathcal{C}_{u,w} + \mathcal{C}_{v,w}$. This way, the correct sum of the distances are accounted for in later stages of the algorithm. The recursion stops as soon as the last hierarchy stage is reached, i.e., the last graph with n' vertices has been partitioned into n'/a_k vertices with a_k vertices each. Recall that vertices in the same block will be assigned to a specified subpart of the system. In this case, a vertex in the graph on the last level of the recursion represents a whole set of tasks with the property that the sum of the vertex weights of each block is precisely the amount of PEs that are present in the subsystem that they are assigned to. We then backtrack the recursion to construct the final mapping.

3.2   Faster Swapping

Initially computing as well as recomputing the objective function after a swap is performed is an expensive step in the algorithm of Brandfass et al. [7]. In their work, both the communication pattern as well as the distances between the PEs are given as complete matrices. These matrices have a quadratic number of elements and hence the initial computation of the objective function costs $O(n^2)$ time. After a swap is performed, Brandfass et al. update the objective using the objective function value before the swap. This is done by looking at all elements in the corresponding columns of the communication and distance matrix. Overall, an update step in their algorithm takes $O(n)$ time which is clearly a bottleneck for sparse communication patterns. We now describe how we speed up the initial computation as well as the update of the objective. As a first step, we rewrite the objective to work with the inverse of the permutation:

$$
\begin{aligned}
J(\mathcal{C}, \mathcal{D}, \Pi) &= \sum_{i,j} \mathcal{C}_{\Pi(i),\Pi(j)} \mathcal{D}_{i,j} \\
&= \sum_{u,v} \mathcal{C}_{u,v} \mathcal{D}_{\Pi^{-1}(u),\Pi^{-1}(v)}
\end{aligned}
$$

with the interpretation that task u is assigned to PE $\Pi^{-1}(u)$. This makes it easier to work with the graph representation of the communication matrix. We rewrite the objective to work with the graph representation instead of the complete communication pattern matrix $\mathcal{C}$:

$$
J(\mathcal{C}, \mathcal{D}, \Pi) := \sum_{(u,v) \in E[\mathcal{C}]} \mathcal{C}_{u,v} \mathcal{D}_{\Pi^{-1}(u),\Pi^{-1}(v)}.
$$

The first observation is that given an initial mapping, we can compute the initial objective in $O(n + m)$ time which is better for sparse graphs. Our next goal is to make the update of the objective fast after a swap has been performed. To do so, let $\Gamma_{\Pi^{-1}}(u) := \sum_{v \in N(u)} \mathcal{C}_{u,v} \mathcal{D}_{\Pi^{-1}(u),\Pi^{-1}(v)}$ be the contribution to the objective of a single vertex u given the current mapping. Note that by using $\Gamma_{\Pi^{-1}}$, we can again rewrite the objective $J(\mathcal{C}, \mathcal{D}, \Pi) := \sum_{u \in V} \Gamma_{\Pi^{-1}}(u)$. Throughout the algorithm, the vertex contributions Γ are always kept up to date. Additionally, it is quite easy to see that performing a swap in the assignment only affects the nodes that are swapped themselves as well as their neighborhood in the communication graph. Hence, we only need to update the node contributions of those nodes and can update the objective accordingly. We update the node contributions as follows: Let u and v be the vertices to be swapped in their assignment Π^{-1}. We start by subtracting the node contributions of all affected nodes from the objective. Before we perform the swap, we iterate over the neighbors of u and v and subtract the contribution induced by the edge connecting the neighbor from its Γ value. We then set the node contributions of u and v to zero and perform the swap. Now we again iterate over all neighbors, basically recomputing the node contributions of u and v, and at the same time adding the new contribution induced by the edge connecting the neighbor to its Γ value. As a last step, we add the new node contributions of all affected nodes from the objective. Overall, this takes $O(d_u + d_v)$ time where d_u and d_v are the degrees of the vertices u and v in the communication graph.

3.3   Alternative Local Search Spaces

We now define swapping neighborhoods using the communication graph $G_\mathcal{C}$. In the simplest version, assignments are only allowed to be swapped if the processes are connected by an edge in the communication graph, i. e., the processes have to communicate with each other.

We denote this neighborhood with $N_{\mathcal{C}}$. The size of the search space is $O(m)$ since it contains exactly m pairs that may be swapped. Swaps are performed in random order. Local search terminates after m unsuccessful swaps, i.e., all pairs have been tried and no swap resulted in a gain in the objective. Note that this approach assumes that swaps with positive gain are close in terms of graph theoretic distance in the communication graph. We also define augmented neighborhoods in which swaps are allowed if two processes have distance less than d in the communication graph. We denote this neighborhood by $N_{\mathcal{C}}^d$. Note that this creates a sequence of neighborhoods increasing in size $N_{\mathcal{C}} \subseteq N_{\mathcal{C}}^2 \subseteq \ldots \subseteq N_{\mathcal{C}}^n = N^2$ where N^2 is the largest neighborhood used by Brandfass et al. [7] (see Section 2). Our experimental section shows that performing swaps with small graph theoretic distance in the communication graph is sufficient to obtain good solutions.

3.4 Miscellanea

Constant Time Distance Oracle. Storing the complete distance matrix requires $O(n^2)$ space. However, due to the problem structure it is not necessary to store the complete matrix. Instead one can build an interval tree over the given PE's describing the hierarchy. The distance of two PEs can then be found by finding the lowest common ancestor in the tree. Such a query can be answered in constant time by investing $O(n)$ preprocessing time [3].

We can use a simpler approach that obtains the distance of two PEs by a few, simple division operations. More precisely, for a hierarchy $\mathcal{S} = a_1, a_2, ..., a_k$ we initially build an array describing the sizes of the intervals on the different levels of the hierarchy. A query then proceeds to scan the implicitly given intervals from top to bottom until the PEs are not on the same subsystem. We then return the corresponding distance.

4 Experiments

Methodology. We have implemented the algorithm described above within the KaHIP framework using C++ and compiled all algorithms using gcc 4.6.3 with full optimization's turned on (-O3 flag). We integrated our algorithms in KaHIP v1.00 graph partitioning framework where the mapping codes are used as post-processing as well as in a separate release VieM [29] (Vienna Mapping and Sparse Quadratic Assignment) to make the mapping codes themselves available to a broader audience. They will integrated into that framework and also released separately. The codes of Brandfass et al. [7] could not be made available to us, so that we implemented those algorithms in our framework as well. Our implementation also uses the sparse representation of the communication pattern. GreedyAllC [15] has been kindly provided by the authors. We also compare against the dual recursive bisection codes of Hofler and Snir [18] (LibTopoMap). Our experiments evaluate the objective of the quadratic assignment problem as well as the running time necessary to compute the solution. To keep the evaluation simple, we use mostly one system hierarchy configuration $\mathcal{D}$. We perform ten repetitions of each algorithm using different random seeds for initialization. Unless otherwise mentioned, we use the geometric mean when reporting averages in order to give every instance the same influence on the *final score*. The system we are using to compute solutions has four Octa-core Intel Xeon E5-4640 processors (32 cores) which run at a clock speed of 2.4 GHz. It has 512 GB local memory.

Instances. We use graphs from various sources to test our algorithm. In Section 4.1, we use these graphs as input to a partitioning algorithm that partitions them into a given number of blocks and then computes the communication graph $\mathcal{C}$ which is the input to our mapping

■ **Table 1** Average running time and average speedup of local search for pruned search space N_p. Here, m/n is the average density of the instances, t_{LS} the average running time of the algorithm using slow gain computations and t_{fastLS} the average running time using fast gain computations.

n	$\overline{m/n}$	$t_{\mathrm{LS}}[s]$	$t_{\mathrm{fastLS}}[s]$	speedup
64	6.7	0.016	0.003	5.3
128	7.3	0.064	0.006	10.7
256	7.9	0.268	0.014	19.1
512	8.3	1.073	0.029	37.0
1K	8.8	4.263	0.059	72.3
2K	9.2	17.083	0.124	137.8
4K	9.7	68.360	0.260	262.9
8K	10.3	268.907	0.540	498.0
16K	11.2	1 075.107	1.158	928.4
32K	12.5	4 348.374	2.472	1 759.1

algorithms. We use the largest six graphs from Chris Walshaw's benchmark archive [30]. Graphs derived from sparse matrices have been taken from the Florida Sparse Matrix Collection [12]. We also use graphs from the 10th DIMACS Implementation Challenge [2] website. Here, `rggX` is a *random geometric graph* with 2^X nodes where nodes represent random points in the unit square and edges connect nodes whose Euclidean distance is below $0.55\sqrt{\ln n/n}$. The graph `delX` is a Delaunay triangulation of 2^X random points in the unit square. The graphs `af_shell9`, `thermal2`, and `nlr` are from the matrix and the numeric section of the DIMACS benchmark set. The graphs `europe` and `deu` are large road networks of Europe and Germany taken from [13]. Basic properties of the graphs under consideration can be found in Table A.3.

4.1   Sparse Quadratic Assignment Problem

In this section, we look at the impact of the various algorithmic components that we presented throughout the paper. In general, we use a hierarchy $\mathcal{S} = a_1, \ldots, a_k$ describing the system hierarchy and communication parameters $D = d_1, \ldots, d_k$ describing the distances between various cores in the subsystems. More precisely, d_i describes the distance of two cores that are in the same subsystems for $i' < i$, and in different subsystems for $i' \geq i$. The total number of cores is given by $n = \prod_i a_i$. Here, we focus on two different system configurations to keep the evaluation simple. Our process in this section is as follows: Take the input graph, partition it into n blocks using the fast configuration of KaHIP, compute the communication graph induced by that (vertices represent blocks, edges are induced by connectivity between blocks, edge cut between two blocks is used as communication volume) and then compute the mapping of the communication graph to the specified system.

4.1.1   Speed-Up of Local Search

We now take the algorithm configurations initially used by Brandfass et al. [7] and investigate the impact of our faster local search algorithms. The configurations are as follows: Use the greedy growing algorithm by Müller-Merbach (as described in Section 2) to provide initial solutions and use the pruned local search neighborhood N_p by Brandfass et al. [7] (see Section 2 for details). We run two configurations: One in which computing the gain takes linear time (the old algorithm) and one with our improved algorithm. In this experiment,
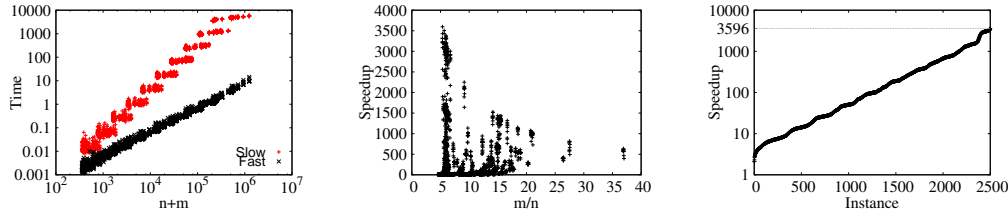
Figure 1 From left to right: Time of local search for both configurations (slow and fast), algorithmic speedup as a function of graph density, algorithmic speedup for the different instances.
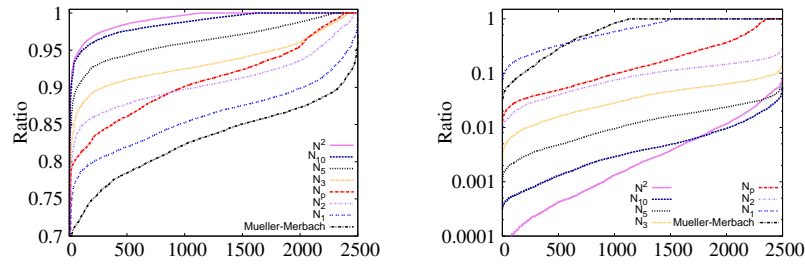
Figure 2 Left/Right: performance plot with respect to solution quality/running time for different local search algorithms.

we use $S = 4 : 16 : k$, $D = 1 : 10 : 100$ with $k = 2^i$, $i \in \{1, ..., 9\}$. Note that the objective of the computed solutions by the algorithm using faster gain computations is precisely the same as their counter part, hence we *do not* report the value of the objective in this section. The results of the experiments are summarized in Figure 1 and Table 1. First, we observe that our new algorithm is *always* faster than the old algorithm. This is expected since the models of computation and communication that are mapped are indeed sparse. Table 1 shows that our fast local search algorithm scales almost linearly in n while the algorithm not using fast gain computations shows quadratic scaling behaviour. The table also already shows a dependency of our algorithm on the density of the instances. This is due to the fact that the gain computation depends on the degrees of the vertices in the communication graph and is in alignment with our theoretical analysis. The expected dependency on the density of the instances can also be seen more clearly in Figure 1. The smallest algorithmic speedup obtained in this experiment is two and the largest speedup is approximately 3 596. We conclude that exploiting the sparsity of the application can improve the running time of local search significantly. From now on, we now always use fast gain computations.

4.1.2   Local Search Neighborhoods

In this section, we look at the influence of local search neighborhoods on final solution quality. The base configuration used here employs the greedy growing algorithm by Müller-Merbach for initialization. Afterwards local search is done using the specified local search neighborhood, i. e., the quadratic neighborhood N^2, the pruned quadratic neighborhood N_p and the communication graph based neighborhoods $N_d := N_C^d$ for $d \in \{1, 2, 3, 5, 10\}$. Again, we use $S = 4 : 16 : k$, $D = 1 : 10 : 100$ with $k = 2^i$, $i \in \{1, ..., 9\}$. To get a visual impression of the solution quality of the different algorithms, Figure 2 presents *performance plots* using all instances. A curve in a performance plot for algorithm X is obtained as

■ **Table 2** Average ratios for solution quality and running time. Baseline denotes the construction heuristic of Müller-Merbach without local search. Algorithms use the baseline algorithm and add local search with the respective local search neighborhood. Comparisons are done against the baseline algorithm. Quality improvements are shown in %.

n	N^2	N_p	N_1	N_2	N_{10}	N^2	N_p	N_1	N_2	N_{10}
	baseline/{baseline+local search}					local search/baseline				
	quality improvement [%]					average running time ratios:				
64	17.4	17.4	6.3	13.0	17.2	26.2	27.1	2.6	13.3	44.1
128	16.0	10.9	3.8	8.5	15.4	63.9	25.2	2.7	16.8	92.8
256	17.3	10.0	3.4	8.3	17.3	114.7	18.9	2.5	16.3	149.0
512	17.6	8.9	3.2	8.0	17.5	171.8	11.3	1.8	12.7	190.2
1K	18.8	8.2	3.1	8.2	18.2	259.1	6.8	1.3	10.0	245.1
2K	19.5	8.1	3.1	8.2	19.1	348.2	3.7	0.9	7.0	258.6
4K	20.5	8.0	3.3	8.7	19.8	472.0	2.0	0.6	5.1	231.8
8K	21.6	8.0	3.6	9.4	20.9	728.2	1.0	0.5	4.0	212.0
16K	23.1	8.3	4.2	10.4	22.1	1 030.8	0.6	0.3	2.9	173.6
32K	25.0	9.1	5.4	11.9	23.7	1 220.9	0.3	0.2	2.1	128.2
overall:	19.68	9.69	3.94	9.46	19.12	443.58	9.69	1.34	9.02	172.54

follows: For each instance, we calculate the ratio between the objective or running time obtained by any of the considered algorithms and objective or running time of algorithm X. These values are then sorted. Additionally, we present average ratios of solution quality and running time in Table 2. First, the local search algorithm using the N_1 neighborhood appears to be the fastest algorithm but also the worst in terms of solution quality. Compared to the initial construction heuristic it takes roughly a factor 1.34 in running time while improving solution quality by roughly 4%. With increasing distance d for the local search neighborhood N_d, solutions improve but also the running time increases. As expected, the local search algorithm using the largest local search neighborhood N^2 computes the best solutions. Here, solutions generated by the initial heuristic are improved by roughly 20%. However, this is also the slowest algorithm (a factor 443 slower than the initial construction heuristic). Also note that we are only able to evaluate the performance of the algorithm at that scale due to the fast gain computations introduced in this paper. Additionally, as n increases the algorithm becomes much slower. This is due to the fact that convergence of the algorithm takes more time for larger n. In contrast, the other local search neighborhoods show much better scaling behaviour as expected. The local search neighborhood N_{10} is faster and computes solutions that are only slightly worse than N^2. For example, for $n = 32K$ the algorithm using N_{10} is more than a factor nine faster and computes solutions that are only 5.5% worse.

4.1.3 Initial Heuristics and Their Scaling Behaviour

We now evaluate the different heuristics that can be used to create solutions. For the evaluation, we employ the algorithm of Müller-Merbach [23], GreedyAllC [15], LibTopoMap [18] (dual recursive bisectioning), Identity, Random, the Bottom-Up as well as the Top-Down and the Top-Down algorithm combined with local search that uses the N_{10} neighborhood (Top-Down+N_{10}). The problems we look at are defined as $S_1 = 4 : 16 : k$, $D_1 = 1 : 10 : 100$ with $k \in \{1, \ldots, 128\}$. We run the Bottom-Up algorithm only to $k \le 50$ due to its large running time. Figure 3 shows the average improvement over solutions obtained by the
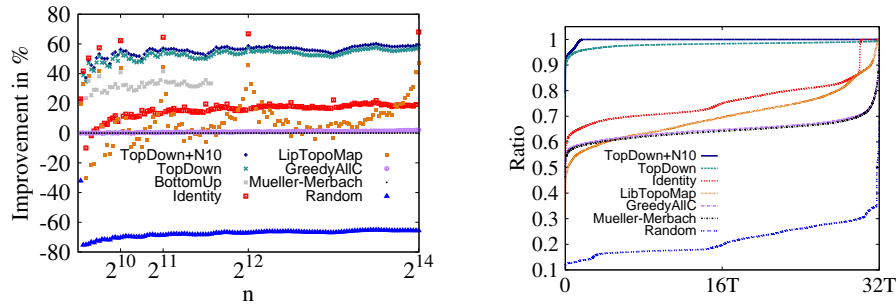
Figure 3 Average improvement in % for different values of n for different algorithms over the algorithm by Müller-Merbach (top) and a performance plot comparing solution quality (bottom).

algorithm of Müller-Merbach and a performance plot for the different algorithms. Indeed, the random mapping algorithms perform worse than the algorithm of Müller-Merbach. On average, the objective computed by the algorithm is 67% worse than the solutions computed by the algorithm of Müller-Merbach. Our Top-Down algorithms yields the best solutions on most of the instances. On average, solutions computed by Top-Down are 52% better than the solutions computed by Müller-Merbach. Adding local search with the N_{10} neighborhood to the algorithm yields additional 5.3% improvement on average. GreedyAllC only improves slightly, i. e., 1% on average, over the algorithm of Müller-Merbach. The identity mapping seems to be the best algorithm for powers of two. This is due to the way the input to the algorithms is constructed, i. e., blocks are initially assigned by KaHIP. To be more precise, KaHIP uses a recursive bisection algorithm on the input graph to compute a model of computation and communication (the input to our mapping algorithms). In each recursion it assigns consecutive blocks to the left side and to the right side. Hence, for powers of two, the identity mapping yields a strategy similar to using recursive bisection on the model to be mapped with good bisections. If the number of elements is not a power of two, then the bisections implied by the identity are not good and hence it performs worse.

LibTopoMap is somewhere in between. It mostly computes better solutions than the greedy algorithms but overall worse solutions than BottomUp and TopDown. On average, solutions are 8% better than the solutions computed by the greedy algorithm of Müller-Merbach. Interestingly, its achieved solution quality is better when the number of vertices in the instances is close to a power of two. This is due to the fact that the algorithm uses dual recursive bisection on the communication and processor graph. However, when the input size is not close to a power of two, there are no good bisections in the processor graph.

In our experiments, Bottom-Up is the slowest algorithm. This is due to the fact that on the coarsest level large partitioning problems have to be solved. The Top-Down algorithm does not have the problem, but is still slower than all other algorithms (except Bottom-Up). On average it is a factor 194 slower than the Müller-Merbach algorithm and a factor 40 slower than GreedyAllC. LibTopoMap is roughly a factor 18 slower than the algorithm of Müller-Merbach. However, the running time of Top-Down is on average only 80% of the time it takes to partition the input graph (using the fast configuration of KaHIP), i. e., the time it takes to create the model which is the input to the mapping algorithms. Adding local search with the N_{10} neighborhood to the algorithm costs additional time, on average 64% of the time it takes to partition the graph. Considering also the high solution quality advantage, we believe that the algorithms are still highly useful in practice.

Scalability.   We now scale the problem size to $n = 2^{19}$ processes/cores. We take the largest graph from our benchmark collection rgg24 and create mapping problems defined as $S_1 = 4 : 16 : 128 : k$, $D_1 = 1 : 10 : 100 : 1000$ with $k \in 2^i, i \in \{1, \ldots, 8\}$. We run Müller-Merbach and the TopDown+N_1 algorithm once. Both algorithms work well on our machine until $i = 4$ ($n = 2^{17}$), at which point there is not sufficient memory available if the implementations use the full distance matrix. Note that the machine has 512GB of memory. Hence, we performed a second run of both algorithms computing distances online (as described in Section 3.4). Note that the version of the Müller-Merbach algorithm is only able to solve larger problem sizes due to both of our changes: the sparse representation of the communication pattern as well as online computation of distances. Computing distances online slows down Müller-Merbach roughly by a factor of five and local search by a factor of three. The running time of TopDown remains the same since it uses the provided hierarchy instead of the distance matrix. In turn the running time advantage of Müller-Merbach also decreases. This is also due to the fact that Müller-Merbachs algorithm is a quadratic time algorithm. For the largest mapping problem ($n = 2^{19}$), the Müller-Merbach algorithm takes a factor 1.64 longer than TopDown. Overall, computing distances online enables a potential user of the algorithms to tackle larger mapping problems.

5   Conclusion

In high performance systems, different cores that are on the same processor usually have the same communication link quality when they communicate with each other, as do cores that are on the same node but not on the same processor and so forth. Using these assumptions, we derived algorithms to create initial mappings as well as faster local search algorithms with alternative local search spaces. Overall, our algorithms drastically speedup local search and are able to compute high quality solutions.

Important future work includes deriving distributed parallel algorithms for the problem. Moreover, we want to investigate algorithms to create a hierarchy of the system if it is not provided as an input to our algorithm. It may be worth to look at more complex local search neighborhoods, e.g., local search spaces that allow to swap whole groups of assignments or allow swapping along cycles in the communication graph. We also want to study the impact of our process mapping on parallel application performance.

References

1   A. H. Abdel-Gawad, M. Thottethodi, and A. Bhatele. RAHTM: Routing Algorithm Aware Hierarchical Task Mapping. In *Int'l Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 325–335, 2014.

2   D. A. Bader, H. Meyerhenke, P. Sanders, C. Schulz, A. Kappes, and D. Wagner. Benchmarking for graph clustering and partitioning. In *Encyclopedia of Social Network Analysis and Mining*, pages 73–82. Springer, 2014.

3   M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Latin American Symposium on Theoretical Informatics*, volume 1776, pages 88–94. Springer, LNCS, 2000.

4   C. Bichot and P. Siarry, editors. *Graph Partitioning*. Wiley, 2011.

5   S. H. Bokhari. On the mapping problem. *IEEE Trans. Computers*, 30(3):207–214, 1981. URL: http://dx.doi.org/10.1109/TC.1981.1675756, doi:10.1109/TC.1981.1675756.

6   S. H. Bokhari. Assignment problems in parallel and distributed computing, 2012.

7   B. Brandfass, T. Alrutz, and T. Gerhold. Rank reordering for MPI communication optimization. *Computers & Fluids*, 80:372–380, 2013.

8 A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. Recent Advances in Graph Partitioning. In *Algorithm Engineering – Selected Topics,* to app*., ArXiv:1311.3144*, 2014.

9 R. E. Burkard, E. Cela, P. M. Pardalos, and L. S. Pitsoulis. The quadratic assignment problem. In *Handbook of combinatorial optimization*, pages 1713–1809. Springer, 1998.

10 Ü. V. Çatalyürek and C. Aykanat. Decomposing Irregularly Sparse Matrices for Parallel Matrix-Vector Multiplication. In *Proc. of the 3rd Int'l Workshop on Parallel Algorithms for Irregularly Structured Problems*, volume 1117, pages 75–86. Springer, 1996.

11 Siew Yin Chan, Teck Chaw Ling, and Eric Aubanel. The impact of heterogeneous multi-core clusters on graph partitioning: an empirical study. *Cluster Computing*, 15(3):281–302, 2012. `doi:10.1007/s10586-012-0229-4`.

12 T. Davis. The University of Florida Sparse Matrix Collection.

13 D. Delling, P. Sanders, D. Schultes, and D. Wagner. Engineering route planning algorithms. In *Algorithmics of Large and Complex Networks*, volume 5515 of *LNCS State-of-the-Art Survey*, pages 117–139. Springer, 2009.

14 J. Fietz, M. Krause, C. Schulz, P. Sanders, and V. Heuveline. Optimized Hybrid Parallel Lattice Boltzmann Fluid Flow Simulations on Complex Geometries. In *Proc. of Euro-Par 2012 Parallel Processing*, volume 7484 of *LNCS*, pages 818–829. Springer, 2012.

15 R. Glantz, H. Meyerhenke, and A. Noe. Algorithms for mapping parallel processes onto grid and torus architectures. In *23rd Euromicro Int'l Conference on Parallel, Distributed, and Network-Based Processing*, pages 236–243. IEEE Computer Society, 2015. `doi:10.1109/PDP.2015.21`.

16 T. Hatazaki. Rank reordering strategy for MPI topology creation functions. In *5th European PVM/MPI User's Group Meeting*, volume 1497 of *LNCS*, pages 188–195. Springer, 1998.

17 C. H. Heider. A computationally simplified pair-exchange algorithm for the quadratic assignment problem. Technical report, DTIC Document, Center for Naval Analyses Arlington VA, 1972.

18 T. Hoefler and M. Snir. Generic topology mapping strategies for large-scale parallel architectures. In *Proc. 25th Int'l Conf. on Supercomputing*, pages 75–84. ACM, 2011.

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

20 G. Mercier and J. Clet-Ortega. Towards an efficient process placement policy for MPI applications in multicore environments. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting (EuroMPI)*, pages 104–115. Springer, 2009.

21 G. Mercier and Emmanuel J. Improving MPI applications performance on multicore clusters with rank reordering. In *18th European MPI Users' Group Meeting*, pages 39–49, 2011.

22 MPI Forum. MPI: A Message-Passing Interface Standard. Version 3.1.

23 H. Müller-Merbach. *Optimale Reihenfolgen*, volume 15 of *Ökonometrie und Unternehmensforschung*. Springer, 1970.

24 P. M. Pardalos and H. Wolkowicz, editors. *Quadratic Assignment and Related Problems, Proceedings of a DIMACS Workshop, New Brunswick, New Jersey, USA, May 20-21, 1993*, volume 16 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. DIMACS/AMS, 1994. URL: `http://dimacs.rutgers.edu/Volumes/Vol16.html`.

25 F. Pellegrini. Scotch Home Page. `http://www.labri.fr/pelegrin/scotch`.

26 S. Sahni and T. F. Gonzalez. P-complete approximation problems. *J. ACM*, 23(3):555–565, 1976. URL: `http://doi.acm.org/10.1145/321958.321975`, `doi:10.1145/321958.321975`.

27 P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *12th Int'l Sym. on Experimental Algorithms (SEA'13)*, volume 7933 of *LNCS*. Springer, 2013.

28 K. Schloegel, G. Karypis, and V. Kumar. Graph Partitioning for High Performance Scientific Simulations. In *The Sourcebook of Parallel Computing*, pages 491–541, 2003.

29 C. Schulz and J. L. Träff. VieM v1.00 – Vienna Mapping and Sparse Quadratic Assignment User Guide. *CoRR*, abs/1703.05509, `http://viem.taa.univie.ac.at/`, 2017. URL: `http://arxiv.org/abs/1703.05509`.

30 A. J. Soper, C. Walshaw, and M. Cross. A Combined Evolutionary Search and Multilevel Optimisation Approach to Graph-Partitioning. *Global Optimization*, 29(2):225–241, 2004.

31 R. V. Southwell. Stress-Calculation in Frameworks by the Method of "Systematic Relaxation of Constraints". *Proc. of the Royal Society of London*, 151(872):56–95, 1935.

32 J. L. Träff. Implementing the MPI process topology mechanism. In *ACM/IEEE Supercomputing*, 2002.

33 J. T. Vogelstein, J. M. Conroy, V. Lyzinski, L. J. Podrazik, S. G. Kratzer, E. T. Harley, D. E. Fishkind, R. J. Vogelstein, and C. E. Priebe. Fast approximate quadratic programming for graph matching, April 2015. `doi:10.1371/journal.pone.0121002`.

34 C. Walshaw and M. Cross. Mesh Partitioning: A Multilevel Balancing and Refinement Algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

35 H. Yu, I-H. Chung, and J. E. Moreira. Topology mapping for Blue Gene/L supercomputer. In *ACM/IEEE Supercomputing*, page 116, 2006.

A   Benchmark Instance Properties

Table 3 Benchmark instance properties.

Graph	n	m
UF Graphs		
cop20k_A	99 843	1 262 244
2cubes_sphere	101 492	772 886
thermomech_TC	102 158	304 700
cfd2	123 440	1 482 229
boneS01	127 224	3 293 964
Dubcova3	146 689	1 744 980
bmwcra_1	148 770	5 247 616
G2_circuit	150 102	288 286
shipsec5	179 860	4 966 618
cont-300	180 895	448 799
Large Walshaw Graphs		
598a	110 971	741 934
fe_ocean	143 437	409 593
144	144 649	1 074 393
wave	156 317	1 059 331
m14b	214 765	1 679 018
auto	448 695	3 314 611
Large Other Graphs		
del23	$\approx$8.4M	$\approx$25.2M
del24	$\approx$16.7M	$\approx$50.3M
rgg23	$\approx$8.4M	$\approx$63.5M
rgg24	$\approx$16.7M	$\approx$132.6M
deu	$\approx$4.4M	$\approx$5.5M
eur	$\approx$18.0M	$\approx$22.2M
af_shell9	$\approx$504K	$\approx$8.5M
thermal2	$\approx$1.2M	$\approx$3.7M
nlr	$\approx$4.2M	$\approx$12.5M