High-Quality Hierarchical Process Mapping

Marcelo Fonseca Faraj
Faculty of Computer Science, University of Vienna, Austria
marcelo.fonseca-faraj@univie.ac.at

Alexander van der Grinten
Humboldt-Universität zu Berlin, Germany
avdgrinten@hu-berlin.de

Henning Meyerhenke
Humboldt-Universität zu Berlin, Germany
meyerhenke@hu-berlin.de

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

Christian Schulz[1]
Faculty of Computer Science, University of Vienna, Austria
christian.schulz@univie.ac.at

Abstract

Partitioning graphs into blocks of roughly equal size such that few edges run between blocks is a frequently needed operation when processing graphs on a parallel computer. When a topology of a distributed system is known, an important task is then to map the blocks of the partition onto the processors such that the overall communication cost is reduced. We present novel multilevel algorithms that integrate graph partitioning and process mapping. Important ingredients of our algorithm include fast label propagation, more localized local search, initial partitioning, as well as a compressed data structure to compute processor distances without storing a distance matrix. Moreover, our algorithms are able to exploit a given hierarchical structure of the distributed system under consideration. Experiments indicate that our algorithms speed up the overall mapping process and, due to the integrated multilevel approach, also find much better solutions in practice. For example, one configuration of our algorithm yields similar solution quality as the previous state-of-the-art in terms of mapping quality for large numbers of partitions while being a factor 9.3 faster. Compared to the currently fastest iterated multilevel mapping algorithm Scotch, we obtain 16% better solutions while investing slightly more running time.

2012 ACM Subject Classification Theory of computation → Design and analysis of algorithms

Keywords and phrases Process Mapping, Graph Partitioning, Algorithm Engineering

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.4

Related Version `http://arxiv.org/abs/2001.07134`

Funding Austrian Science Fund (FWF, project P 31763-N31); DFG grant FINCA (ME-3619/3-2, SPP 1736 Algorithms for Big Data); German Federal Ministry of Education and Research (BMBF, project WAVE, grant 01|H15004B).

---

[1] Corresponding author.

© Marcelo Fonseca Faraj, Alexander van der Grinten, Henning Meyerhenke, Jesper Larsson Träff, and Christian Schulz;
licensed under Creative Commons License CC-BY
18th International Symposium on Experimental Algorithms (SEA 2020).
Editors: Simone Faro and Domenico Cantone; Article No. 4; pp. 4:1–4:15

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

The performance of applications that run on high-performance computing systems depends on many factors such as the capability and topology of the underlying communication system, the required communication between processes in the given applications, and the software and algorithms used to realize the communication. For example, communication is typically faster if communicating processes are located on the same physical processor node compared to cases where processes reside on different nodes. This becomes even more pronounced for large supercomputer systems where processing elements are hierarchically organized (e.g., islands, racks, nodes, processors, cores) with corresponding communication links of similar quality, and where differences in the process placement can have a huge impact on the communication performance (latency, bandwidth, congestion). Often the communication pattern between application processes is or can be known. Additionally, a hardware topology description that reflects the capacity of the communication links is typically available. Hence, 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 are located closely. Another typical application of process mapping can be found in data allocation [33] problems where expensive optimization algorithms are used to assign jobs to data centers in order to minimize expected wait times. Finding such best or just good mappings is the objective of some usually hard optimization problems.

Previous work can be grouped into two categories. One line of research intertwines process mapping with multilevel graph partitioning (see for example [37, 20]). To this end, the objective of the partitioning algorithm – most commonly the number of cut edges – is typically replaced by an objective function that considers the processor distances. Throughout these algorithms, the distances are directly taken into consideration. The second category decouples partitioning and mapping (see for example [29, 3, 12, 18]). First, a graph partitioning algorithm is used to partition a large graph into k blocks, while minimizing some measure of communication, such as edge-cut, and at the same time balancing the load (size of the blocks). Afterwards, a coarser model of computation and communication is created in which the number of nodes matches the number of processing elements (PEs) in the given processor network. This model is then mapped to a processor network of k PEs with given pair-wise distances.

Recently, process mapping algorithms have made two assumptions that are typically valid for modern supercomputers and the applications that run on those: communication patterns are sparse and there is a hierarchical communication topology where links on the same level in the hierarchy exhibit the same communication speed. Using these assumptions, better non-integrated mapping algorithms have been obtained [29]. Here, the model of computation and communication is first partitioned using a standard graph partitioning algorithm, and then a smaller model that has the same number of nodes as the underlying network of processors is mapped. On the other hand, there has been a large body of work on the multilevel (hyper-)graph partitioning problem, which led to enhanced partitioning quality or faster local search [24, 25, 17, 27, 26]. The *multilevel* approach [4] is probably the most prominently used algorithm. Here, the input is recursively *contracted* to obtain a smaller instance which should reflect the same basic structure as the input. After applying an *initial partitioning* algorithm to the smallest instance, contraction is undone and, at each level, *local search* methods are used to improve the partitioning induced by the coarser level.

Our *main contribution* in this paper is the integration of process mapping into a multilevel scheme with high-quality local search techniques and recently developed non-integrated mapping algorithms. Additionally, we introduce faster techniques that avoid to store

distance matrices. The rest of this paper is organized as follows. In Section 2, we introduce basic concepts and describe relevant related work in more detail. We present our main contributions in Section 3. We present a summary of extensive experiments to evaluate algorithm performance in Section 4. The experiments indicate that our new integrated algorithm improves mapping quality over other state-of-the-art integrated and non-integrated mapping algorithms. For example, one configuration of our algorithm yields similar solution quality as the previous state-of-the-art in terms of mapping quality for large values of k while being a factor 9.3 faster. Compared to the currently fastest iterated multilevel mapping algorithm Scotch, we obtain 16% better solutions while investing slightly more running time. Most importantly, hierarchical multisection algorithms that take the system hierarchy into account for model creation improve the results of the overall process mapping significantly.

2 Preliminaries

The communication requirements between the components of a set of processes in (some section of) an application can be represented by a weighted communication graph. The underlying hardware topology can likewise be represented by a weighted graph, particularly a complete graph since any two physical processors can communicate with each other facilitated by the routing system. This complete graph can be represented by a topology cost matrix reflecting the costs of routing along shortest or cheapest paths between physical processors. Furthermore, it does not need to be explicitly expressed if the topology is organized as a regular hierarchy of components with fixed communication cost per message inside each level. We tackle the problem of embedding a communication graph onto a topology graph under optimization criteria that we explain below. Unless otherwise mentioned, a processing element (PE) represents a core of a machine.

2.1 Basic Concepts

Let $G = (V = \{0, \ldots, n-1\}, E)$ be an *undirected graph* with edge weights $\omega : E \to \mathbb{R}_{>0}$, vertex weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We generalize c and ω functions to sets, such that $c(V') = \sum_{v \in V'} c(v)$ and $\omega(E') = \sum_{e \in E'} \omega(e)$. Let $\Gamma(v) = \{u : \{v, u\} \in E\}$ denote the neighbors of a vertex v. Let $I(v)$ denote the set of edges incident to v. A graph $S = (V', E')$ is said to be a *subgraph* of $G = (V, E)$ if $V' \subseteq V$ and $E' \subseteq E \cap (V' \times V')$. When $E' = E \cap (V' \times V')$, S is an *induced* subgraph.

The *graph partitioning problem* (GPP) consists of assigning each node of G to exactly one of k distinct blocks respecting a balancing constraint in order to minimize the edge-cut. More precisely, GPP partitions V into k blocks $V_1, \ldots, V_k$ (i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$), which is called a *k-partition* of G. The *balancing constraint* demands that the sum of node weights in each block does not exceed a threshold associated with some allowed *imbalance* ϵ. More specifically, $\forall i \in \{1, \ldots, k\} : c(V_i) \leq L_{\max} := \lceil (1+\epsilon)\frac{c(V)}{k} \rceil$. Let a block V_i be called *λ-underloaded* if $c(V_i) + \lambda \leq L_{\max}$ and *overloaded* if $c(V_i) > L_{\max}$. The *edge-cut* of a k-partition consists of the total weight of the edges crossing blocks, i.e., $\sum_{i<j} \omega(E_{ij})$, where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$. An abstract view of the partitioned graph is a *quotient graph* $\mathcal{Q}$, in which nodes represent blocks and edges are induced by the connectivity between blocks. More precisely, there is an edge in the quotient graph if there is an edge that runs between the blocks in the original, partitioned graph. We call *neighboring blocks* a pair of blocks connected to each other by an edge in the quotient graph. If a node $v \in V_i$ has a neighbor $w \in V_j, i \neq j$, then it is called a *boundary* node. Let $R(v)$ be the set of all blocks containing at least one element from $\{v\} \cup \Gamma(v)$.

Assume that we have n processes and a topology containing k PEs. Let $\mathcal{C} \in \mathbb{R}^{n\times n}$ denote the communication matrix and let $\mathcal{D} \in \mathbb{R}^{k\times k}$ denote the (implicit) topology matrix or distance matrix. In particular, $\mathcal{C}_{i,j}$ represents the required amount of communication between processes i and j, while $\mathcal{D}_{x,y}$ represents the cost of each communication between PEs x and y. Hence, if processes i and j are respectively assigned to PEs x and y, or vice-versa, the communication cost between i and j will be $\mathcal{C}_{i,j}\mathcal{D}_{x,y}$. Throughout this work, we assume that $\mathcal{C}$ and $\mathcal{D}$ are symmetric – otherwise one can create equivalent problems with symmetric inputs [3].

In this work, we deal with topologies organized as homogeneous hierarchies, even though our algorithms could be extended to heterogeneous hierarchies in a straightforward way. Let $\mathcal{S} = a_1 : a_2 : ... : a_\ell$ be a sequence describing the hierarchy of a supercomputer. The sequence should be interpreted as each processor having a_1 cores, each node a_2 processors, each rack a_3 nodes, and so forth, such that the total number of PEs is $k = \Pi_{i=1}^{\ell} a_i$. Let $D = d_1 : d_2 : \ldots : d_\ell$ be a sequence describing the communication cost inside each hierarchy level, meaning that two cores in the same processor communicate with cost d_1, two cores in the same node but in different processors communicate with cost d_2, two cores in the same rack but in different nodes communicate with cost d_3, and so forth.

Throughout the paper, 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 weight of each edge in the graph equals the corresponding entry in the communication matrix $\mathcal{C}$.

Our *main focus* in this work is the *general process mapping problem* (GPMP). It consists of assigning each node of a given communication graph to a specific PE in a communication topology while respecting a balancing constraint (the same as in the graph partitioning problem above) in order to minimize the total communication costs. Within the scope of this work, the number of nodes (processes) n in the communication graph is much larger than the number of PEs k in the topology graph which matches most real-world situations. Let the mapping function that maps a node onto its block be $\Pi : \{1, \ldots, n\} \mapsto \{1, \ldots, k\}$. Hence, the objective function of GPMP is to minimize $J(\mathcal{C}, \mathcal{D}, \Pi) := \sum_{i,j} \mathcal{C}_{i,j}\mathcal{D}_{\Pi(i),\Pi(j)}$. Many authors deal with the specific case in which $n = k$, resulting in the *one-to-one process mapping problem* (OPMP), where each process i is assigned to a unique PE $\Pi(i)$. Within the context of OPMP, searching for the inverse permutation instead, i.e., assigning PE x to node $\Pi^{-1}(x)$, results in the same problem since Π is a bijection.

GPP and OPMP are both NP-hard problems [8, 22]. Since GPP and OPMP are special cases of GPMP, the latter is also NP-hard. Two of the most common methods to solve GPMP are the two-phase approach and the integrated approach. In the *two-phase* approach, GPMP is solved in two consecutive steps: (i) a heuristic for GPP is applied in the communication graph, obtaining a balanced k-partition; (ii) a heuristic for OPMP is used to map the blocks of the k-partition onto the topology of PEs. On the other hand, the *integrated* approach consists of tackling GPMP directly, i.e., not decomposing the input problem into k independent sub-problems first.

2.2 Multilevel Approach

In this section, we characterize the multilevel approach within the scope of GPMP, although the same basic structure is extensible to many other problems, such as GPP. Before describing the multilevel scheme, we need to define the terms contraction and uncontraction. *Contracting*

an edge $e = \{u, v\}$ consists of replacing the nodes u and v by a new node x connected to the former neighbors of u and v and setting $c(x) = c(u) + c(v)$. If replacing edges of the form $\{u, w\}$, $\{v, w\}$ would generate two parallel edges $\{x, w\}$, a single edge with $\omega(\{x, w\}) = \omega(\{u, w\}) + \omega(\{v, w\})$ is inserted. *Uncontracting* undoes contraction.

A *multilevel approach* to solve GPMP consists of three main phases. In the *contraction* (coarsening) phase, successive approximations of an original input graph are created. The contractions quickly reduce the size of the graph and stop as soon as it becomes sufficiently small to be partitioned and mapped by an expensive algorithm. In the construction phase, an initial mapping is produced for the coarsest approximation of the input graph. Due to the way we define contraction, every mapping of the coarsest level implies a corresponding mapping of the input graph with equal objective function and balance. In the *local improvement* (or uncoarsening) phase, we uncontract previously contracted nodes to go back through each level, from the coarsest approximation to the original graph. After each uncoarsening, local improvement algorithms move nodes between blocks in order to improve the objective function or balance.

2.3 Related Work

There has been an immense amount of research on GPP – we refer to [2, 4, 28] for extensive material. The most successful general-purpose methods to solve GPP for huge real-world graphs are based on the multilevel approach. The most commonly used formulation of the multilevel scheme was proposed in [13]. Among the most successful multilevel software packages to solve GPP, we mention Jostle [36], Metis [14], Scotch [19], and KaHIP [23].

Systems like KaHIP [23] and Metis [14] typically compute a k-partition on the coarsest level through a recursive bisection strategy or a direct k-way partitioning scheme. In recursive bisection, the graph is recursively divided into two blocks until the number of blocks is reached, i.e., a bisection algorithm is used to split the graph into two blocks. More precisely, each bisection step itself uses a multilevel algorithm. To obtain a bipartition in the coarsest level, KaHIP uses the *greedy graph growing* algorithm. In KaHIP, if k is not even, the graph gets split into two blocks, V_1 and V_2, such that $c(V_1) \leq \lfloor \frac{k}{2} \rfloor L_{\max}$, $c(V_2) \leq \lceil \frac{k}{2} \rceil L_{\max}$. Block V_1 will be recursively partitioned in $\lfloor \frac{k}{2} \rfloor$ blocks and block V_2 will be recursively partitioned in $\lceil \frac{k}{2} \rceil$ blocks.

In addition to GPP, Jostle and Scotch can also solve GPMP. Jostle integrates local search into a multilevel scheme to partition the model of computation and communication. In this scheme, it solves the problem on the coarsest level and afterwards performs refinements based on the user-supplied network communication model. Scotch performs dual recursive bipartitioning to compute a mapping. More precisely, it starts the recursion considering all given processes and PEs. At each recursion level, it bipartitions the communication graph and also the distance graph with a graph bipartitioning algorithm. The first (resp., second) block of the communication graph is then assigned to the first (resp., second) block of the distance graph.

There is likewise a large literature on OPMP, often in the context of scientific applications using the *Message Passing Interface* (MPI). Hatazaki [11] was among the first authors to propose graph partitioning to solve the MPI process mapping of a virtual unweighted topology onto a hardware topology organized in modules and sub-modules. In [31], a similar approach was used to implement one of the first non-trivial mappings designed for the NEC SX-series of parallel vector computers. In [15] and later [16], the mapping problem was simplified to ignore the whole network topology except that inside each node. These works also investigated multiple placement policies to enhance overall system performance. In [34],

the authors proposed a gradient–based heuristic for OPMP that involves solving assignment problems, and gave experimental evidence for better solution quality and speed compared to other heuristics.

Müller-Merbach [18] proposed a greedy construction method to obtain an initial permutation for OPMP. The method roughly works as follows: Initially compute the total communication volume for each process and also the sum of distances from each core to all the others. Afterwards, the algorithm proceeds in rounds. In each round, the process with the largest communication volume is assigned to the core with the smallest total distance. Glantz et al. [9] noted that the algorithm does not link the choices for the vertices and cores and hence propose a modification of this algorithm called *GreedyAllC* (the best algorithm in [9]). GreedyAllC links the mapping choices by scaling the distance with the amount of communication to be done.

A method to improve an already given solution for OPMP was proposed in [12]. The method repeatedly tries to perform swaps in the assignment in 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 $\Pi^{-1}(i)$ will be assigned to processor j and $\Pi^{-1}(j)$ will be assigned to processor i after the swap is done. The algorithm then looks at the neighborhood in a cyclic manner. A swap is performed if it reduces the objective. To reduce the runtime, Brandfass et al. [3] introduced a couple of modifications to speed up the algorithm, such as only considering pairs for swapping that can reduce the objective or partitioning the search space into s consecutive blocks and only performing swaps inside those blocks.

In [29], GPMP was tackled with a two-phase approach. First, the graph is partitioned using KaHIP (which uses recursive bisection). Their best OPMP algorithm, *hierarchy top down*, recursively partitions the communication graph into blocks defined by the given hierarchy. A local search similar to that from [3] with faster data structures was also used to improve the initially computed mapping. The authors experimentally showed that N_C^{10}, which restricts swapping to processes that have a distance smaller than 10 in the communication graph, is an adequate choice to obtain good solutions with a moderate running time.

The OPMP algorithm proposed in [10] requires the hardware topology to be a partial cube, i. e. an isometric subgraph of a hypercube. This requirement allows to label (i) the PEs as well as (ii) the nodes of the application graph G with meaningful bit-strings along convex cuts. These bit-strings facilitate (i) the fast computation of distances between PEs and (ii) an effective hierarchical local search method to improve the mapping induced by the labels. Subsequently, in [35], the graph partitioning step was modified to already use *hierarchical multisection* itself. This yields better communication graphs for the second OPMP mapping step.

3 High-Quality Multilevel General Process Mapping

We engineered all the components of a multilevel algorithm to solve GPMP in an *integrated* way, as illustrated in Figure 1. In this section, we present our algorithmic contributions and discuss each of their components. This includes coarsening-uncoarsening schemes, methods to obtain initial solutions, local refinement methods, and additional tools to explore trade-offs in memory usage and performance.

3.1 Coarsening

We use a matching-based coarsening scheme. The *matching–based* coarsening is the most common choice in multilevel partitioning algorithms due to its simplicity, speed, and generality. It has two consecutive steps: An edge rating function and a matching algorithm. Based on local information, the *edge rating function* scores each edge to estimate the benefit of contracting it. We employ the same edge rating function $\exp^*(e) = \omega(e)/(|\Gamma(u)| * |\Gamma(v)|)$ as used in [24]. Then, the *matching algorithm* obtains a maximal match to maximize the sum of the ratings of the contracted edges. As in [24], we computed matchings with the *Global Paths Algorithm* [24], which is a $\frac{1}{2}$-approximate algorithm.

3.2 Initial Solution Algorithms

We compute the initial mapping using a two-phase approach. To solve GPP, we compare two multilevel recursive bisection algorithms: (i) *standard bisection* setup, in which we perform a recursive bisection to obtain k blocks; (ii) *multisection* setup, in which we perform recursive bisections throughout the hierarchical structure of PEs. To construct a solution for OPMP, we apply two different construction methods: (i) *identity*, which assigns each block to the PE with the same ID to favor locality; (ii) *hierarchy top down*, which partitions the set of blocks throughout the hierarchical structure of PEs. To refine the OPMP solution, we perform an $N_{\mathcal{C}}^{10}$ swap neighborhood local search. Hence, the resulting map Π of nodes to PEs becomes our initial GPMP solution.

Our *standard bisection* setup for initial partition corresponds to the initial partition step in KaHIP. Moreover, it is a canonical choice to produce initial solutions in multilevel schemes tackling GPP. On the other hand, the *multisection* setup draws inspiration from the scheme used in [35]. It is an attempt to specialize the initial partition for the particular case tackled in this paper: a regularly hierarchical distribution of PEs in which the communication cost between two processes (nodes) highly depends on the hierarchy level shared by their corresponding PEs (blocks). Particularly, we apply a recursive partitioning scheme that splits all the nodes in a_ℓ blocks, then splits the nodes in each block in $a_{\ell-1}$ sub-blocks, then splits the nodes in each sub-blocks in $a_{\ell-2}$ sub-sub-blocks, and so forth. Observing that the communication costs decrease as the communicating processes share lower hierarchy levels, the multisection approach implies a hierarchy of sub-problems that directly reflects the problem cost hierarchy.

In both setups of the partitioning step, we recursively assign consecutive IDs to blocks throughout the process in order to maintain locality. Moreover, the PEs belonging to each hierarchy module are labeled with consecutive IDs, which also promotes locality. Then, the

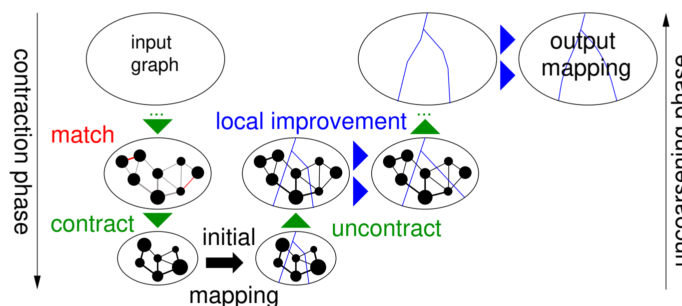


Figure 1 Multilevel scheme used to solve GPMP (Figure from [24]).

identity method is a fast way to construct a solution for OPMP taking advantage of this locality: it assigns each block V_i to the PE with the ID i. Note, the *standard bisection* setup conveniently combines with the identity mapping approach when k is a power of 2 since the recursive bisections will be automatically performed throughout the hierarchical topology. For an analogous reason, the *multisection* setup is a good algorithm to create a coarse model to be mapped by the *identity* mapping approach independently of k. The *hierarchy top down* [29] is a more general approach to construct solutions for OPMP when the PEs are hierarchically organized. Its mechanism is similar to the idea of multisection throughout the hierarchy.

3.3 Uncoarsening

After obtaining an initial solution for GPMP at the coarsest level, we apply a sequence of four local refinement methods to move nodes between blocks (which are already associated to unique PEs). Then, we undo each of the contractions performed previously, from the coarsest graph until the original input graph. After each uncoarsening step, we repeat our four local refinement methods. The refinements run in a specific order based on their characteristics. First, a *quotient graph refinement* exhaustively tries to improve solution quality and eliminate imbalance by moving nodes between each pair of blocks connected by an edge in the quotient graph. Second, a *k-way Fiduccia-Mattheyses (FM) algorithm [7, 32] refinement* greedily goes through the boundary nodes trying to relocate them with a more global perspective in order to improve the mapping. Third, a *label propagation refinement* randomly visits all nodes and moves each one to the most appropriate block while not increasing the objective. Finally, a *multi-try FM refinement* is exhaustively applied in rounds with random starting points throughout the graph in order to escape local optima as many times as possible. Before explaining the local search algorithms, we introduce the notion of *gain* for GPMP.

Gain. All our refinement methods are based on the concept of *gain*. We define $\Psi_b(v)$ as the *partial* contribution of a node v to the objective function $J(\mathcal{C}, \mathcal{D}, \Pi)$ in case v is assigned to the PE b. More precisely, $\Psi_b(v)$ represents the total cost of the communications involving v if $\Pi(v) = b$ and the neighbors of v remain assigned to their current PEs. The *gain* $g_b(v)$ represents the value that will be subtracted from $J(\mathcal{C}, \mathcal{D}, \Pi)$ if a node v is moved from its current PE $\Pi(v)$ to PE b. More precisely, $\Psi_b(v) := \sum_{\{v,u\} \in I(v)} \mathcal{C}_{v,u} \mathcal{D}_{b,\Pi(u)}$ and $g_b(v) := \Psi_{\Pi(v)}(v) - \Psi_b(v)$. Note that $g_{\Pi(v)}(v) \equiv 0$. Observe that a positive (resp., negative) gain indicates improvement (resp., worsening) of the solution. Computing the gains of v to all blocks in $R(v)$ costs $O\big(|R(v)||I(v)|\big) = O\big(|I(v)|^2\big)$. For comparison purposes, the computation of the same corresponding gains in the context of GPP and edge-cut objective function costs $O\big(|I(v)|\big)$.

Quotient Graph Refinement. We implemented an adapted version of the *quotient graph refinement* [24] to incorporate our definition of gains. Within this refinement, we visit each pair of neighboring blocks in the quotient graph $\mathcal{Q}$ underlying the current k-partition. Then we apply an FM algorithm [7] to move nodes between the two currently visited blocks, keeping two respective gain–based priority queues of eligible nodes. Each queue is randomly initialized with the boundary in its corresponding block. After a node is moved (which can only happen once during an execution of the local search), its unmoved neighbors become eligible.

K-Way FM Refinement. Our k-way FM refinement was adapted from the implementation in [24]. Unlike the quotient graph refinement, the k-way FM does not restrict the movement of a node to a certain pair of blocks, but performs global-aware movement choices. Our implementation of k-way FM uses only one gain–based priority queue P, which is initialized with the *complete* partition boundary in a random order. Then, the local search repeatedly looks for the highest-gain node v and moves it to the best $c(v)$-underloaded neighboring block. When a node is moved, we insert in P all its neighbors that were not in P and have not been moved yet. The k-way local search stops if P is empty (i.e., each node was moved once) or when a stopping criterion based on a random-walk model described in [24] applies. To escape from local optima, this refinement allows some movements with negative gain or to blocks that are not $c(v)$-underloaded. Afterwards local search is rolled back to the lowest cut fulfilling the balance criterion that occurred.

Label Propagation Refinement. We propose a local search inspired by *label propagation* [21]. The algorithm works in rounds. In each round, the algorithm visits all nodes in a random order, starting with the labels being the current assignment of nodes to blocks. When a node v is visited, it is moved to the $c(v)$-underloaded neighboring block with highest positive gain. We consider only $c(v)$-underloaded blocks since this ensures that the target block is not overloaded when the node is moved there. Ties are broken randomly and a 0-gain neighboring block can be occasionally chosen with 50% probability if there is no neighboring $c(v)$-underloaded block with positive gain. We perform at most ℓ rounds of the algorithm, where ℓ is a tuning parameter.

Multi-Try FM Refinement. We also adapted our gain concept to a localized variant of the k-way local search algorithm similar to that proposed in [24] under the name of *multi-try* FM. Instead of being initialized with all boundary nodes, as in k-way FM, multi-try FM is repeatedly initialized with a single boundary node. This introduces a higher diversification to the search since it is not restricted to movements in boundary nodes with global largest gain. As a result, this local search can escape local optima more easily than *k-way* FM.

3.4 Additional Techniques

Implicit Distance Matrix. When the topology matrix $\mathcal{D}$ is stored in memory, access time to obtain the distance between a pair of PEs is $O(1)$, but this requires $O(k^2)$ space. From now on, we refer to the algorithm explicitly keeping $\mathcal{D}$ in memory as *matrix–based* approach. We implement three alternative approaches to save memory by exploiting the fact that our topology matrix is a hierarchy and the IDs of PEs in each of the hierarchy modules are sequential. For simplification reasons, we call these approaches: (i) *division–based*; (ii) *stored division–based*; and (iii) *binary notation–based*.

In the *division–based* approach, we perform $O(\ell)$ successive integer divisions and comparisons in the ID of two PEs when we need to find out their distance. Here, ℓ is the number of levels in the system hierarchy. As a preprocessing step to be executed only once, we create a vector $h = \left(k \Big/ \prod_{t=1}^{\ell} a_t,\ k \Big/ \prod_{t=2}^{\ell} a_t,\ \ldots,\ k \Big/ a_\ell\right)$. To find the distance between PEs b and b' with $b \neq b'$, we loop through the hierarchy layers from $i = \ell$ to $i = 1$. In each iteration, we perform the integer division of b and b' by h_i. Whenever the division results differ, then we break the loop and return $\mathcal{D}_{b,b'} = d_i$. This approach does not require any additional memory other than a vector with $O(\ell)$ integers and has time complexity $O(\ell)$.

The *stored division–based* approach works in a similar way as the *division–based* one. The only difference is that we avoid repetitive integer divisions of IDs by elements of h by storing the results of all possible divisions in a preprocessing step executed only once. Although we still need $O(\ell)$ running time to perform comparisons in order to obtain the distance between a pair of PEs, the constant factors involved are much lower. This improvement in running time comes at the cost of additional $O(k\ell)$ memory.

The *binary notation–based* approach is a more compact way of decomposing the IDs of PEs. Instead of storing ℓ numbers for each PE, we keep in memory a single binary number per PE. This binary number r consists of ℓ sections r_i, each containing $s = \lceil \log_2 \left(\max_{1 \leq t \leq \ell}(a_t) \right) \rceil$ bits. To describe the construction of r for a PE b, let a variable t be initialized as $t = b$. Then, we loop through the hierarchy layers, from $i = 1$ to $i = \ell$. In each iteration i, r_i receives the remainder of the division of t by a_i and, then, t is updated to store the integer quotient of t by a_i. Afterwards, it is possible to precisely locate b at the hierarchy by sweeping the sections of r from r_ℓ to r_1. In particular, r_ℓ specifies its data center, $r_{\ell-1}$ specifies its server among those belonging to its data center, and so forth. Obtaining the distance between distinct PEs b and b' is equivalent to finding which section r_i contains the leftmost nonzero bit in the result of the bit-wise operation XOR(b,b'). The running-time complexity of finding the section of the leftmost nonzero bit is $O(\log(\ell))$. Furthermore, current processors often implement a *count leading zeros* (CLZ) operation in hardware which allows the identification of the leftmost nonzero bit in $O(1)$ time, under the assumption that the size $\log r = O(\log k)$ of the binary numbers is smaller than the size of a machine word.

Delta-Gain Updates. Our local searches frequently need to compute *gains* involved in the movement of nodes. A *base approach* to check these gains consists of computing them from scratch whenever they are needed, which can yield many gain recomputations. For this reason, we implement a technique to save running time called *delta-gain updates* [26].

In *delta-gain updates*, we store a vector R of length $|R(v)| = O(|\Gamma(v)|)$ for each node v. In this vector, we keep the gains $g_b(v)$ for all PEs b containing neighbors of v. Additionally, we store an n-sized vector h to keep flags that indicate whether a node has up-to-date gains in memory. Asymptotically speaking, these vectors represent $O(n + m)$ extra memory. Each flag is initialized with an inactive seed and is considered active if its value equals a counter that is increased after each uncoarsening steps. When we need to check a gain of some node v, we look at h_v to verify if the gains of v are up-to-date. If they are not, we compute all gains $g_b(v)$ from scratch, which costs $O\big(|I(v)|^2\big)$, and activate h_v. Otherwise, we just access the required gain from memory in $O(1)$ time.

If a node v moves from its current PE to another one, we have to update all delta gains of v and $u \in \Gamma(v)$ with h_u being active. Assume that h_v and h_u are active and v moves from PE 1 to PE 2 during some local refinement. After this movement, we should change the delta gains of u and v in memory. For v, it suffices to subtract $g_2(v)$ from all other gains of v and then set $g_2(v)$ to 0. For u, it is slightly trickier, but we do not need to recalculate all its gains from scratch since their only source of change is the edge e that connects u and v. Hence, we respectively subtract and add to $g_b(u)$ the corresponding contribution of e before and after the movement of v. We end up doing the update in time $O\big(|I(v)| + |I(v)| * \overline{|R(u)|}\big)$, where $\overline{|R(u)|}$ is the average of $|R(u)|, \forall \{v, u\} \in I(v)$.

4 Experimental Evaluation

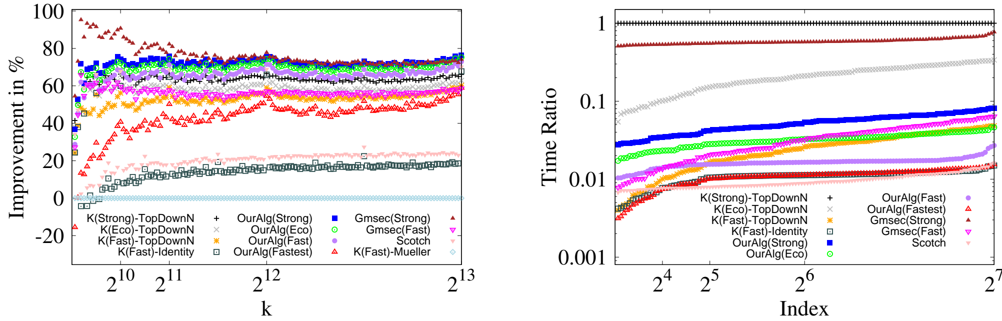
Methodology. We performed our implementations using the KaHIP framework (using C++) and compiled them using gcc 8.3 with full optimization turned on (-O3 flag). All of our experiments were run on a single core of a machine with four sixteen-core Intel Xeon Haswell-EX E7-8867 processors running at 2.5 GHz, 1 TB of main memory, and 32768 KB of L2-Cache. The machine runs Debian GNU/Linux 10 and Linux kernel version 4.19.67-2.

For experiments based on the two-phase approach for tackling GPMP, we solve GPP using KaHIP [24]. We use its configurations *fast*, *eco* and *strong* which are described in [24] – we respectively refer to them as *K(Fast)*, *K(Eco)* and *K(Strong)*. KaHIP also contains the *top down* approach to solve OPMP. We also run Scotch [19] configured to only use recursive bipartitioning methods using the quality setting. We contacted Christopher Walshaw, who informed us that Jostle [36] is not available anymore. Lastly, we compare against global multisection [35] in which the graph partitioning step is already using *hierarchical multisection* itself. We use the configurations *Gmsec(Strong)*, *Gmsec(Eco)* and *Gmsec(Fast)* which differ in the configuration used for partitioning in a global multisection way.

To keep the evaluation simple, we use the following hierarchy configurations for all the experiments: $D = 1 : 10 : 100$, $\mathcal{S} = 4 : 16 : r$, with $r \in \{1, 2, 3, \ldots, 128\}$. Hence, $k = 64 \cdot r$. Depending on the focus of the experiment, we measure running time and/or $J(\mathcal{C}, \mathcal{D}, \Pi)$. We perform ten repetitions of each algorithm using different random seeds for initialization and calculate the arithmetic average of the computed objective functions and running time. When further averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the *final score*. Some of our plots are performance profiles. These plots relate the running times of all algorithms to the slowest algorithm on a per-instance basis. For each algorithm, these ratios are sorted in increasing order. The plots show $\left(\frac{\sigma_{\text{A}}}{\sigma_{\text{slowest}}}\right)$ on the y-axis. A point close to zero indicates that the algorithm was considerably faster than the slowest algorithm.

Instances. Our instances come from various sources. We use the largest six graphs from Chris Walshaw's benchmark archive [30]. Graphs derived from sparse matrices have been taken from the SuiteSparse Matrix Collection [5]. We also use graphs from the 10th DIMACS Implementation Challenge [1] website. Basic properties of the graphs under consideration can be found in Table 1 of the full version of the paper [6].

Algorithm Configuration. We performed a wide range of experiments to tune our algorithm (on the tuning graphs from (Table 1, [6]). Due to space constraints, we refer the reader to the full version of the paper [6] for details. After the tuning step, we have four configurations of our algorithm: All of our algorithms use the binary notation algorithm to compute distances between PEs as this is favorable over storing the distance matrix in terms of speed and memory (see [6]). (i) *fast* uses multisection to compute initial solutions without additional OPMP local search, label propagation with delta-gain updates; (ii) *eco* computes initial solutions as the fast configuration and uses OPMP local search, quotient graph refinement, k-way FM, label propagation; and (iii) *strong* applies multisection to compute initial solutions with additional OPMP local search, and uses all available local search methods. To improve speed even more, we include a configuration called *fastest* which is the same as the fast configuration but does not use local search during uncoarsening.



(a) Improvements in objective function over K(Fast)-Müller-Merbach. Higher is better.

(b) Performance profile for running time. Lower is better.

Figure 2 Comparisons against state-of-the-art approaches for GPMP.

Comparison with State of the Art. In this section, we compare our algorithms against the best alternative algorithms in the literature. We report experiments on all graphs listed in (Table 1, [6]) – excluding the graph used to tune our algorithm. We select the most successful algorithms from [29] and also Scotch for our comparison: (i) *Top down* with N_C^d local search (TopDownN), which represent the state-of-the-art for OPMP when k is not a power of 2; (ii) *identity* mapping, which (when coupled with the KaHIP multilevel partitioning algorithm) represents the state-of-the-art for GPMP via two-phase approach when k is a power of 2; (iii) the algorithm of *Müller-Merbach* [18] (Müller-Merbach), whose results are also used as a reference algorithm to calculate solution improvements in [29]; and (iv) *Scotch* [19]. We run the two-phase approches TopDownN, Identity, and Müller-Merbach coupled with K(Fast), K(Eco) and K(Strong) as a partitioning algorithm. We also compare against global multisection [35], in which the graph partitioning step is already using *hierarchical multisection* itself. Additionally, we use the algorithms Gmsec(Strong), Gmsec(Eco) and Gmsec(Fast). Recall that these algorithms are non-integrated: they use different quality configurations of KaHIP to partition the graph, compute the coarser communication model, and then apply TopDownN to solve OPMP. Scotch is among the algorithms with best running times in our experiments. Hence, we add an algorithm (ScotchTC) which reports the best solution out of multiple runs of Scotch with different random seeds when given the same amount of time to compute a solution as our *strong* configuration has used. Figure 2 gives an overview over our results.

Overall, Scotch has the lowest average running time, directly followed by our algorithm *fastest*, K(Fast)-Identity, and our algorithm *fast* (respectively 9%, 10%, and 73% slower than Scotch on average). Next, the average running time of K(Fast)-TopDownN and Gmsec(Fast) are respectivelly a factor 2.3 and 3.1 higher than Scotch. For our algorithms *eco* and *strong*, this factor is respectively 3.3 and 5.4. By definition ScotchTC is also a factor 5.4 higher than Scotch. Next, Gmsec(Eco) and K(Eco)-TopDownN have much higher running times (9.3 and 20.4 times slower than Scotch, respectively). Finally, Gmsec(Strong) and K(Strong)-TopDownN are the slowest ones (62 and 107 times slower than Scotch, respectively).

We now highlight the comparison of various configurations/algorithms. Gmsec(Strong) is the algorithm with best overall mapping quality. It is 2.5% on average better compared to our *strong* configuration, however our *strong* configuration is more than an order of magnitude faster on average – a factor 11.5 faster. Better quality of Gmsec(Strong) stems from the fact

that global multisection itself already takes the system hierarchy into account and hence yields good models to be mapped. Moreover, the graph partitioning approach itself which is used to compute a communication graph also uses more (time-consuming) sophisticated local search algorithms. This includes methods such as flow-based methods which particularly work well for small values of k as well as global search methods such as V-cycles. In particular for $k > 2^{11}$, our *strong* has the same average quality as Gmsec(Strong) but is 9.3 times faster. Our algorithm computes similar solutions in much less time since the multilevel algorithm directly optimizes the correct objective. For $k > 2^{11}$, our *eco* is 2.4% better and 2.8 times faster than Gmsec(Eco), and our *fast* is 9% better and 2.6 times faster than Gmsec(Fast). Overall, our *strong* configuration improves solution quality over K(Strong)-TopDownN by 5.1% while being a factor 20 on average faster. Our *eco* configuration has roughly 3.6% better quality than K(Strong)-TopDown but is a factor 32 faster on average. Our *fast* configuration still yields 1.3% better solutions than K(Strong)-TopDown on average, and is a factor 62 faster. Here, improvements stem from the fact that the new algorithms are integrated and not two-phase as well as the fact that these algorithms do not perform multisections. Lastly, our *fastest* algorithm is on average 9% slower than Scotch but also improves solution quality over Scotch by 16%. Our *strong* algorithm is 40% better than Scotch and consumes a factor 5.4 more running time.

5 Conclusion

We tackled the general process mapping problem, which is to assign a larger set of processes to a smaller set of processing elements such that total communication cost between cores is minimzed. Our algorithms integrate graph partitioning and process mapping. Important ingredients of our algorithm include fast label propagation, more localized local search, initial partitioning, as well as a compressed data structure to compute processor distances without storing a distance matrix.

Experimental results indicate that our algorithms are the new state-of-the-art for general process mapping. In particular, our algorithms generate much better or similar overall solutions in comparison to any of the competitors while being an order of magnitude faster than the previous best algorithm in terms of quality. Our improvements are mostly due to the integrated multilevel approach combined with high-quality local search algorithms and initial mapping algorithms that split the initial network along the specified system hierarchy.

Important future work includes parallelization as well as the integration of global search schemes and different types of coarsening to improve solution quality further. Moreover, we plan to investigate the impact on the real performance of applications such as sparse matrix vector multiplications. Lastly, we plan to release the proposed algorithms in the VieM (`http://viem.taa.univie.ac.at/`) and KaHIP (`http://algo2.iti.kit.edu/kahip/`) frameworks.

References

1. 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.
2. C. Bichot and P. Siarry, editors. *Graph Partitioning.* Wiley, 2011.
3. B. Brandfass, T. Alrutz, and T. Gerhold. Rank reordering for MPI communication optimization. *Computers & Fluids*, 80:372–380, 2013.
4. A. Buluç, H. Meyerhenke, I. Safro, P. Sanders, and C. Schulz. *Recent Advances in Graph Partitioning*, pages 117–158. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-49487-6_4.
5. T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, 2011. doi:10.1145/2049662.2049663.
6. M. Fonseca Faraj, A. van der Grinten, H. Meyerhenke, J. L. Träff, and C. Schulz. High-quality hierarchical process mapping. *CoRR*, abs/2001.07134, 2020. arXiv:2001.07134.
7. C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *Proc. of the 19th Conference on Design Automation*, pages 175–181, 1982.
8. M. R. Garey, D. S. Johnson, and L. Stockmeyer. Some Simplified NP-Complete Problems. In *Proc. of the 6th ACM Symposium on Theory of Computing*, (STOC), pages 47–63. ACM, 1974.
9. R. Glantz, H. Meyerhenke, and A. Noe. Algorithms for mapping parallel processes onto grid and torus architectures. In *23rd Euromicro Intl. Conference on Parallel, Distributed, and Network-Based Processing*, pages 236–243, 2015.
10. R. Glantz, M. Predari, and H. Meyerhenke. Topology-induced enhancement of mappings. In *Proceedings of the 47th International Conference on Parallel Processing, ICPP 2018*, pages 9:1–9:10. ACM, 2018. doi:10.1145/3225058.3225117.
11. 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, 1998.
12. C. H. Heider. A computationally simplified pair-exchange algorithm for the quadratic assignment problem. Technical report, DTIC Document, 1972.
13. B. Hendrickson and R. Leland. A Multilevel Algorithm for Partitioning Graphs. In *Proc. of the ACM/IEEE Conference on Supercomputing'95*. ACM, 1995.
14. 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.
15. G. Mercier and J. Clet-Ortega. Towards an efficient process placement policy for MPI applications in multicore environments. In *16th European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, volume 5759 of *LNCS*, pages 104–115. Springer, 2009.
16. G. Mercier and E. Jeannot. Improving MPI applications performance on multicore clusters with rank reordering. In *18th European MPI Users' Group Meeting*, volume 6960 of *LNCS*, pages 39–49. Springer, 2011.
17. H. Meyerhenke, P. Sanders, and C. Schulz. Partitioning Complex Networks via Size-constrained Clustering. In *13th Int. Symp. on Exp. Algorithms*, volume 8504 of *LNCS*. Springer, 2014.
18. H. Müller-Merbach. *Optimale reihenfolgen*, volume 15 of *Ökonometrie und Unternehmensforschung.* Springer-Verlag, 1970.
19. F. Pellegrini. Scotch Home Page. http://www.labri.fr/pelegrin/scotch.
20. François Pellegrini and Jean Roman. SCOTCH: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs. In *High-Performance Computing and Networking*, volume 1067 of *Lecture Notes in Computer Science*, pages 493–498. Springer, 1996. doi:10.1007/3-540-61142-8_588.
21. U. N. Raghavan, R. Albert, and S. Kumara. Near linear time algorithm to detect community structures in large-scale networks. *Physical Review E*, 76(3):036106, 2007.
22. S. Sahni and T. F. Gonzalez. P-complete approximation problems. *J. ACM*, 23(3):555–565, 1976. doi:10.1145/321958.321975.

23 P. Sanders and C. Schulz. KaHIP – Karlsruhe High Qualtity Partitioning Homepage. `http://algo2.iti.kit.edu/documents/kahip/index.html`.

24 P. Sanders and C. Schulz. Engineering Multilevel Graph Partitioning Algorithms. In *Proc. of the 19th European Symp. on Algorithms*, volume 6942 of *LNCS*, pages 469–480. Springer, 2011.

25 P. Sanders and C. Schulz. Think Locally, Act Globally: Highly Balanced Graph Partitioning. In *12th Intl. Sym. on Experimental Algorithms (SEA)*, LNCS. Springer, 2013.

26 S. Schlag, V. Henne, T. Heuer, H. Meyerhenke, P. Sanders, and C. Schulz. k-way hypergraph partitioning via n-level recursive bisection. In *Proceedings of the Eighteenth Workshop on Algorithm Engineering and Experiments, ALENEX*, pages 53–67, 2016. `doi:10.1137/1.9781611974317.5`.

27 C. Schulz. *High Quality Graph Partititioning*. PhD thesis, Karlsruhe Institute of Technology, 2013.

28 C. Schulz and D. Strash. Graph partitioning: Formulations and applications to big data. In Sherif Sakr and Albert Y. Zomaya, editors, *Encyclopedia of Big Data Technologies*. Springer, 2019. `doi:10.1007/978-3-319-63962-8_312-2`.

29 C. Schulz and J. L. Träff. Better process mapping and sparse quadratic assignment. In *16th International Symposium on Experimental Algorithms*, volume 75 of *LIPIcs*, pages 4:1–4:15, 2017. `doi:10.4230/LIPIcs.SEA.2017.4`.

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 J. L. Träff. Implementing the MPI process topology mechanism. In *ACM/IEEE Supercomputing*, pages 40:1–40:14, 2002.

32 Jesper Larsson Träff. Direct graph k-partitioning with a Kernighan-Lin like heuristic. *Operations Research Letters*, 34(6):621–629, 2006.

33 R. Vamosi, M. Lassnig, and E. Schikuta. Data allocation based on evolutionary data popularity clustering. In Yong Shi, Haohuan Fu, Yingjie Tian, Valeria V. Krzhizhanovskaya, Michael Harold Lees, Jack J. Dongarra, and Peter M. A. Sloot, editors, *Computational Science - ICCS 2018 - 18th International Conference, Wuxi, China, June 11-13, 2018, Proceedings, Part I*, volume 10860 of *Lecture Notes in Computer Science*, pages 153–166. Springer, 2018. `doi:10.1007/978-3-319-93698-7_12`.

34 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. PLOS One, April 2015.

35 K. von Kirchbach, C. Schulz, and J. L. Träff. Better process mapping and sparse quadratic assignment. *CoRR*, 2019. URL: `http://arxiv.org/abs/1702.04164v2`.

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

37 C. Walshaw and M. Cross. Multilevel mesh partitioning for heterogeneous communication networks. *Future Generation Comp. Syst.*, 17(5):601–623, 2001. `doi:10.1016/S0167-739X(00)00107-2`.