An MPI-based Algorithm for Mapping Complex Networks onto Hierarchical Architectures [*]

Maria Predari[1], Charilaos Tzovas[1], Christian Schulz[2], and
Henning Meyerhenke[1]

[1] Humboldt-Universität zu Berlin, Berlin, Germany
{charilat,predarim,meyerhenke}@hu-berlin.de
[2] Universität Heidelberg, Heidelberg, Germany
christian.schulz@informatik.uni-heidelberg.de

Abstract. Processing massive application graphs on distributed memory systems requires to map the graphs onto the system's processing elements (PEs). This task becomes all the more important when PEs have non-uniform communication costs or the input is highly irregular. Typically, mapping is addressed using partitioning, in a two-step approach or an integrated one. Parallel partitioning tools do exist; yet, corresponding mapping algorithms or their public implementations all have major sequential parts or other severe scaling limitations.
In this paper, we propose a parallel algorithm that maps graphs onto the PEs of a hierarchical system. Our solution integrates partitioning and mapping; it models the system hierarchy in a concise way as an implicit labeled tree. The vertices of the application graph are labeled as well, and these vertex labels induce the mapping. The mapping optimization follows the basic idea of parallel label propagation, but we tailor the gain computations of label changes to quickly account for the induced communication costs. Our MPI-based code is the first public implementation of a parallel graph mapping algorithm; to this end, we extend the partitioning library ParHIP. To evaluate our algorithm's implementation, we perform comparative experiments with complex networks in the million- and billion-scale range. In general our mapping tool shows good scalability on up to a few thousand PEs. Compared to other MPI-based competitors, our algorithm achieves the best speed to quality trade-off and our quality results are even better than non-parallel mapping tools.

Keywords: load balancing, process mapping, hierarchical architectures, parallel label propagation

1 Introduction

Task mapping is the process of assigning tasks of a parallel application onto a number of available processing elements (PEs) and is an important step in

high-performance computing. One reason for the importance of mapping is non-uniform memory access (NUMA), common in many modern architectures, where PEs close to each other communicate faster than PEs further away. The importance stems from the fact that communication is orders of magnitude slower than computation. To alleviate those issues, task mapping is often used as a preprocessing step. Successful mapping solutions assign pairs of heavily-communicating tasks "close to each other" in the parallel system, so that their communication overhead is reduced. Moreover, the network topologies of parallel architetures exhibit special properties that can be exploited during mapping. A common property is that PEs are hierarchically organized into, e. g., islands, racks, nodes, processors, cores with corresponding communication links of similar quality.

Furthermore, mapping becomes even more important when the application's structure and communication pattern are highly irregular. While partitioning the application may work well for mesh-based numerical simulations, large graphs derived from social and other complex networks pose additional challenges [34]. Typical examples are power-law [3] and small-world graphs [21]. The former are characterized by highly skewed vertex degree distribution, the latter exhibit a particularly low graph diameter. Distributed graph processing systems, such as Apache Giraph and GraphLab [23], are made to run analytics on such graphs. For some algorithms, in particular those with local data access, they report good scaling results [23]. For non-local or otherwise complex analytics and highly irregular inputs, running times and scalability of these systems can become unsatisfactory [30]. Consequently, designing MPI-based graph processing applications is necessary to scale to massive instances with high performance. Considering the above and the number of PEs in modern architectures (a number expected to increase in the future), task mapping can have significant impact in the overall application performance [1, 4]. Good mapping algorithms should be able to improve the quality of the final mapping, and additionally, they need to be fast in order not to degrade the overall performance of the application. Thus, developing MPI-based mapping algorithms with good scaling behavior becomes all the more crucial. Since finding optimal topology mappings is NP-hard [14], heuristics are often used to obtain fairly good solutions within reasonable time [5, 35].

Our contribution is an MPI-based, integrated mapping algorithm for hierarchically organized architectures, implemented within ParHIP. To model the system hierarchy and the corresponding communication costs, we use an implicit bit-label representation, which is very concise and effective. Our algorithm, called ParHIP_map, uses parallel label propagation to stir the mapping optimization. As far as we know ParHIP_map is currently the only available implementation for parallel mapping. Experiments show that it offers the best speed to quality trade-off; having on average 62% higher quality than the second best competitor (ParHIP), and being only 18% slower than the fastest competitor (xtraPulp), which favors speed over quality. Moreover, our algorithm scales well up to 3 072 PEs and is able to handle graphs of billion edges, with the least failing rate among other MPI-based tools due to memory or timeout issues on massive com-

plex networks. Moreover, compared to a sequential baseline mapping algorithm, PARHIP_MAP has on average 10% better quality results.

2 Background

We model the underlying parallel application with a graph $G_a = (V_a, E_a, \omega_a)$, where vertex u_a represents a computational task of the application and an edge e_a indicates data exchanges between tasks. The amount of exchanged data is quantified by the edge weight $\omega_a(e_a)$. Network information for hierarchically organized systems is often modeled with trees[3] [13, 19]; we do so as well, but implicitly (see Sec. 3). The bottom-up input description of the topology follows KAHIP: $H = \{h_0, h_1, \ldots, h_{l-1}\}$ denotes the number of children of a node per level, where l is the number of hierarchy levels; i. e., each processor has h_0 cores, each node h_1 processors, and so on. We also define the set of PEs as V_p of size $k = \prod_{i=0}^{l-1} h_i$. Communication costs are defined via $D = \{d_0, d_1, \ldots, d_{l-1}\}$ such that PEs with a common ancestor in level i of the hierarchy communicate with cost d_i; i. e., two cores in the same processor communicate with d_0 cost, two cores in the same node but in different processors with d_1, and so on. We use $d(u_p, v_p)$ to indicate the time for one data unit exchange between PEs u_p and v_p (i. e., their distance).

A k-way partition of G divides V into k blocks $V_1, \ldots, V_k$, such that $V_1 \cup \ldots \cup V_k = V$, $V_i \neq \emptyset$ for $i = 1, \ldots, k$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. Graph partitioning aims at finding a k-way partition of G that optimizes an objective function while adhering to a balance constraint. The balance constraint demands that the sum of node weights in each block does not exceed a given imbalance threshold ϵ. Moreover, the objective function is often taken to be the edge-cut of the partition $\sum_{i<j} w(E_{ij})$, where $E_{ij} := \{\{u, v\} \in E : u \in V_i, v \in V_j\}$.

A mapping, $\mu : V_a \mapsto V_p$, is defined as a nearly balanced assignment of computational tasks onto PEs such that, for some imbalance parameter $\varepsilon \geq 0$, $|\mu^{-1}(v_p)| \leq (1 + \varepsilon) \cdot \lceil |V_a|/|\mu(V_a)| \rceil$ for all $v_p \in \mu(V_a)$. Hence, $\mu(\cdot)$ induces a balanced partition of G_a with blocks $\mu^{-1}(v_p)$, $v_p \in V_p$. Or inversely, a mapping μ defines a one-to-one mapping from k balanced blocks of V_a to k PEs of V_p. To steer an optimization process for obtaining good mappings, different objective functions have been proposed [14].[4] A widely used [27] mapping objective is $\mathrm{Coco}(\cdot)$ (also referred to as *hop-byte* or *qap*), defined as:

$$\mathrm{Coco}(\mu) := \sum_{\substack{e_a \in E_a \\ e_a = \{u_a, v_a\}}} \omega_a(e_a)\ d(\mu(u_a), \mu(v_a)) \tag{1}$$

Intuitively speaking, placing pairs of highly communicating tasks in nearby PEs minimizes $\mathrm{Coco}(\cdot)$.

---

[3] In a tree topology, leaf vertices correspond to PEs, internal nodes to switches.

[4] Most theoretical metrics can only approximate the communication overhead of the application since communication during real-time execution can be affected by many external factors (e.g., network traffic and overhead from competing jobs).

2.1 Related Work

In this section we focus on related techniques for parallel graph partitioning and sequential and parallel task mapping. For more details we refer the reader to the overview articles for task mapping [15, 27] and for graph partitioning [7].

Parallel graph partitioning. Graph partitioning is closely related to task mapping. First, because it is often used as a building block for mapping and second, because it substitutes mapping when no network information is available. A trivial mapping can be computed from the solution of a graph partitioner, simply by assigning block i to PE i (identity mapping). To this end, a graph-based metric, such as the edgecut, *i. e.*, the total weight of edges between blocks, is minimized. Some popular parallel partitioners are ParMETIS [33], ParHIP [24], and PT-Scotch [28]. These tools all follow the multilevel framework, performing one or more cycles of the following procedure: they construct a hierarchy of successively coarser graphs, find an initial solution on the coarsest graph and project the solution successively to the original graph, while refining it on every level. Graph coarsening is often based on edge matching [33] or label propagation [29], while initial partitioning uses recursive bisection, local heuristics or evolutionary algorithms. The main bottleneck for high scalability in these tools seems to be the high memory usage due to successive coarsening and the poor scaling of the initial partitioning phase. xtraPulp [34], a single-level parallel partitioning tool that uses label propagation, avoids the scalability issue. This advantage comes with the price of reduced quality, though.

Mapping tools. Existing mapping algorithms are grouped into two categories: integrated approaches and two-phase ones. Integrated approaches address the mapping problem using the network information directly, without decomposing the problem into independent sub-problems. Examples of integrated approaches are included in Scotch [26] and KaHIP [32]. Scotch uses dual recursive bisection (DRB) [25] to partition both the application graph and the network topology into two blocks recursively. Embedded sectioning [20] and Recursive multisection [8,16] follow a similar technique. Recently, Faraj *et al.* [10] proposed an integrated mapping approach that uses fast label propagation and a more localized local search to achieve mapping solutions of high quality. LibTopoMap [14], TopoMatch [17] and MpiPP [9] are typical examples of the two-phase approach, where mapping is solved in two steps. The first step usually involves an established partitioner that obtains a balanced partition. The second step then assigns the resulting blocks to the PEs while minimizing a mapping objective, *e. g.* using a greedy approach [6, 14] or metaheuristics [6, 19].

   To the best of our knowledge, all current mapping algorithms or their public implementations have major sequential parts. LibTopoMap and TopoMatch use parallel partitioning, but the mapping step is completely sequential. Regarding integrated approaches, PT-Scotch offers parallel mapping only for the trivial case where the underlying network topology corresponds to a complete

graph, which is simply partitioning.[5] Moreover, the JOSTLE authors briefly discuss a parallel mapping extension of their sequential approach but do not provide enough details nor an implementation [36].

2.2   Parallel Label Propagation with Size Constraints

Our mapping approach uses the parallel label propagation algorithm (LPA) with size constraints [24], as implemented in PARHIP. We discuss the algorithm and its implementation here for self-containment reasons. In its sequential form, LPA starts with some partition (depending on the algorithm's purpose) and iterates over all vertices. At each vertex v, the block number (= label) of v is set to the dominant one in the neighborhood of v (= block with highest total edge weight incident to v). If a size constraint is imposed, then the dominant block that can still host v is chosen. The loop over all vertices vertices is repeated, *e. g.* a fixed number of times or until no more changes occur.

The parallel version is implemented as follows. First, each PE gets a subgraph of the input graph consisting of a contiguous range of nodes in the interval $I := [a \dots b]$, the edges incident to the nodes of those blocks, as well as the end points of edges which are not in I (so-called ghost or halo nodes). In any case, the graph data structure only stores edges incident to local vertices. To parallelize the label propagation algorithm, each PE visits all local vertices in a random order. A vertex v is moved to the block that has the strongest eligible connection such that the block will not be overloaded. During the course of the algorithm, local vertices can change their block and hence the blocks in which halo vertices are contained can change as well. Communication is expensive, so instead of updating labels of halo vertices every time they change, the algorithm follows an overlapping scheme, organized in phases. A node is called interface node if it is adjacent to at least one ghost node. The PE associated with the ghost node is called adjacent PE. Each PE stores a separate send buffer for all adjacent PEs. During each phase, the algorithm stores the block label of interface nodes that have changed into the send buffer of each adjacent PE of this node. Communication is then implemented asynchronously. In phase i, the current updates are sent to the adjacent PEs and each PE receives the updates of the adjacent PEs from round $i - 1$, for $i > 1$.

For maintaining the weight of blocks, the algorithm uses two different approaches, one for coarsening and another for uncoarsening. During coarsening, the algorithm uses a localized approach for keeping up with the block weight since the number of blocks is high and the balance constraint is not tight. Roughly speaking, a PE maintains and updates only the local amount of node weight of the blocks of its local and ghost nodes. Due to the way the label propagation algorithm is initialized, each PE knows the exact weights of the blocks of local nodes and ghost nodes in the beginning. Label propagation then uses the local

---

[5] In the documentation of PT-SCOTCH (6.0.1), there is a comment about implementing parallel mapping algorithms for more target architectures in future releases.

information to bound the block weights. Once a node changes its block, the local block weight is updated. This does not involve additional communication.

During uncoarsening a different approach is taken compared to coarsening since the number of blocks is much smaller. This bookkeeping approach is similar to the one in PARMETIS [33]. Initially, the exact block weights of all k blocks are computed locally. The local block weights are then aggregated and broadcast to all PEs. Both can be done using one allreduce operation. Now each PE knows the global block weights of all k blocks. The label propagation algorithm then uses this information and locally updates the weights. For each block, a PE maintains and updates the total amount of node weight that local nodes contribute to the block weights. Using this information, one can restore the exact block weights with one allreduce operation which is done at the end of each computation phase.

3   Parallel Mapping Algorithm

In this section, we present the main technical contributions of the paper. This includes an integrated mapping algorithm for distributed memory systems based on a concise representation of the hierarchical network topology via bit-labels. Our algorithm uses a parallel local search refinement process, where gain computations for label changes account for communication in the network topology. The bit-label network representation allows a quick gain evaluation. As far as we know, this is the first (implemented and publicly available) parallel mapping algorithm for distributed-memory systems.

3.1   Network Topology Representation

To encode a tree network topology, most representations typically store l numbers for each PE, one for each tree hierarchy level. However, in our work we use a concise bit-label representation that has very low space requirement. For each vertex $v \in V_p$ (*i. e.*, each PE), we only store a single number as a bit-label. This number is also hidden within the labels of vertices in V_a as a bit-prefix. This enables us to use label propagation on G_a for the refinement process and quickly evaluate distances between PEs.

The bit-label of a given vertex $v_p \in V_p$ encodes the full ancestry of v_p in the tree. The ancestor of v_p in level i can be viewed as a block of an implicit partition in that level (see Figure 1). The local numberings of all ancestors/blocks of v_p are encoded in the bit-label of v_p and indicate ownership of the vertex in the tree hierarchy (in which rack, processor, node etc., it belongs to). The red vertex in Figure 1 has a label of 131 (10|000|0|11 in binary). Reading the bit-label from left to right, we have that v belongs to block 2 of the first level partition, block 0 of the second level and so on. For the network topology this means that PE v belongs to rack 2, local processor 0, local node 0 and has local core number 3. To construct the labels for the available PEs we use Algorithm 1. The bit-label of each PE is divided into l sections, each containing $s[i]$ bits with $0 \leq i < l$, as in Line 5. Each subsection r_i gives the local numbering of the ancestor node of
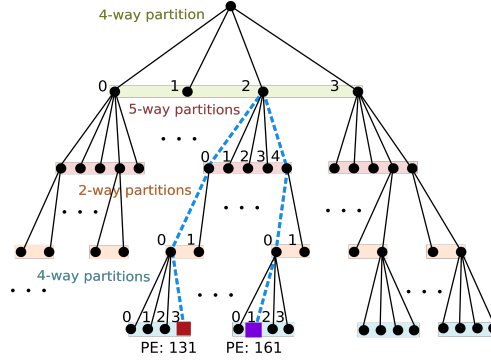
Fig. 1: Example of implicit tree topology. The system has 160 PEs in one island with four racks (first level), five nodes per rack (second), two processors per node (third) and four cores per processor (forth). PEs take labels from 0 to 231 but not all labels are used due to different number of children per level.

the current PE in level i. For each PE p we loop through the levels of the tree hierarchy and encode the local numbering of each ancestor node in the bit-label of p (see Lines 9 to 13).

To retrieve the distance between two PEs, one needs to find their common ancestor in the tree topology. To achieve this in our representation, we apply the bit-wise operation $xor(\cdot, \cdot)$ on the two bit-labels. We find the level of the common ancestor by finding the section r_i which contains the leftmost non-zero bit on the result of $xor(\cdot, \cdot)$. In the example of Figure 1, the leftmost non-zero bit of the squared vertices is in the second section. This corresponds to the second level in the tree (illustrated with the blue dashed line). The time complexity of finding the section of the leftmost non-zero bit is $\mathcal{O}(\log l)$. Note that modern processors often have hardware implementation of a count leading zeros operation. This makes the identification of the leftmost non-zero bit a constant time operation.[6]

3.2   Refinement & Gain Computation

Our parallel mapping approach is an integrated solution method that performs one cycle of the multilevel framework. More precisely, we coarsen the graph, compute an initial partition and uncoarsen the solution while refining with local search based on a mapping objective such as Equation (1). In a parallel setting, each graph vertex v_a has a local vertex label, within the PE, and a global one. The global vertex labels can be used to induce a mapping $\mu(\cdot)$ onto PEs via their prefixes. For instance, in Figure 2 all vertices of the shaded block of G_a have labels with prefix 00|01|01, implying a mapping to PE 5. Using the concise network representation, retrieving communication costs and evaluating Coco can be

---

[6] This holds under the realistic assumption that for any bit-label v_p, the size $\log v_p = \mathcal{O}(\log k)$ of the binary numbers is smaller than the size of a machine word.

Algorithm 1 Algorithm for building the bit-label representation for PEs.

```
1: function BUILDLABEL(H)
2:     Input: H, i. e., number of children per node for each hierarchy level
3:     Output: x, i. e., array of bit-label representation for PEs
4:     l ← size(H)                              ▷ number of hierarchies in the tree topology
5:     s[i] ← ⌈log₂ hᵢ⌉                                            ▷ array of size l
6:     k ← ∏ᵢ₌₁ˡ hᵢ
7:     for p ← 0 to k − 1 do
8:         t ← p
9:         for i ← 0 to l − 1 do
10:            r[i] ← t mod hᵢ
11:            t ← t/hᵢ
12:            x[i] ← r[i] << (i ∗ s[i])
13:        end for
14:    end for
15:    return x
16: end function
```

---

performed quickly for all edges of G_a. For an edge $e_a = (u_a, v_a) \in E_a$, the prefixes of u_a and v_a are used to compute the communication costs between $\mu(u_a)$ and $\mu(v_a)$, by returning the leftmost non-zero bit on the result of $xor(u_a, v_a)$. Overall, through the bit-label information, we can quickly refine an initial mapping using parallel label propagation. We use the size-constrainted version of the algorithm and tailor the gain computations of a potential vertex move to account for the induced communication costs.

The optimization process works as follows: all PEs visit their local vertices in random order and consider moving a given vertex v into another block from the set of candidates $R(v) = \{\mu(u) : u \in N(v)\} \subseteq V_p$ ($N(v)$ is the neigborhood of v). The algorithm performs the move that induces the maximum improvement in Coco as long as the size constraint is respected. To compute the best block assignment, we temporarily move v to each block in $R(v)$ and calculate the communication cost for all possible block assignments; finally, we set $\mu(v) := \operatorname{argmin}_{b \in R(v)} \left(\sum_{u \in N(v)} w(v, u) \, d_{G_p}(b, \mu(u)) \right)$. If the maximum reduction is induced by keeping the vertex in the current block, we do not perform any move. In Figure 2a and for an implicit tree distance of $D = \{2, 4, 10\}$, the maximum reduction for vertex v, with label $00|01|01|***$. After a certain number of moves, we do a global communication step to update the labels of the halo nodes of each PE. To handle overloaded blocks, we keep the same modifications to the block selection rule that was proposed in [24]. The process is repeated for a user-defined number of rounds. This is repeated for each refinement level and on the original graph.

3.3   Overall Approach

Here we present a more detailed description of the fully parallel mapping algorithm designed for distributed-memory systems. We implement our algorithm in PARHIP. For the coarsening step we use the parallel size-constrained label propagation as implemented in PARHIP, without any modifications. Each PE computes clusters of its local graph and aggregates them in super-vertices in
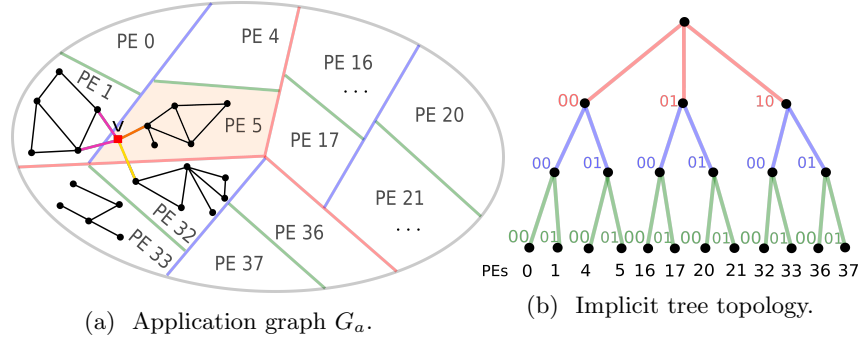
(a) Application graph G_a.

(b) Implicit tree topology.

Fig. 2: Mapping from G_a to the (implicit) tree topology: colored lines in (a) indicate the cuts induced by the tree levels in (b). The decimal PE numbers in (a) correspond to the combined bit-labels on a path from the root to the PE leaf in (b).

parallel until the coarsened graph becomes small enough. The distributed coarse graph is then collected on each PE and is partitioned using a min-cut/max-flow algorithm within an evolutionary framework [31]. The best solution among all PEs is kept and broadcast back to them. For the uncoarsening step each PE performs a local search using the parallel size-constrained label propagation on their local part of the coarse graph. This step is repeated for each level of the hierarchy to refine the solution. At this point, we adapt the parallel label propagation to account for the communication overhead among PEs. More precisely, we modify the block selection step during vertex moving and we use the implicit topology representation to quickly retrieve the distance costs among PEs. During gain computation we change the objective from edgecut to $\text{Coco}(\cdot)$.

Avoiding memory issues. As already mentioned in Section 2.1, classic multilevel algorithms have high memory usage caused by multiple cycles of successive coarsening. Our algorithm performs only one cycle of the multilevel framework, but successive coarsening (even in one cycle) can still damage the scalability of our approach. Moreover, the previous implementation of PARHIP (without our contributions for parallel mapping) uses block partitioning for the initial data distribution.[7] Block partitioning of the input graph often leads to many inter-PE edges even before coarsening, in particular for complex networks. One reason for that is the high-degree nodes of complex networks, also known as hubs, when they become halo vertices. When the number of hubs being halo vertices becomes large enough, the scalability of PARHIP is negatively affected.

In our implementation we propose a simple correction for such instances to avoid high memory usage: first, all PEs globally identify halo nodes of high degree (halo hubs) and then each PE temporarily removes edges connecting halo hubs with non-local nodes, creating a reduced local graph. Then each PE runs

---

[7] The initial data distribution is not a mapping solution, only an initial assignment of the input data to the PEs.
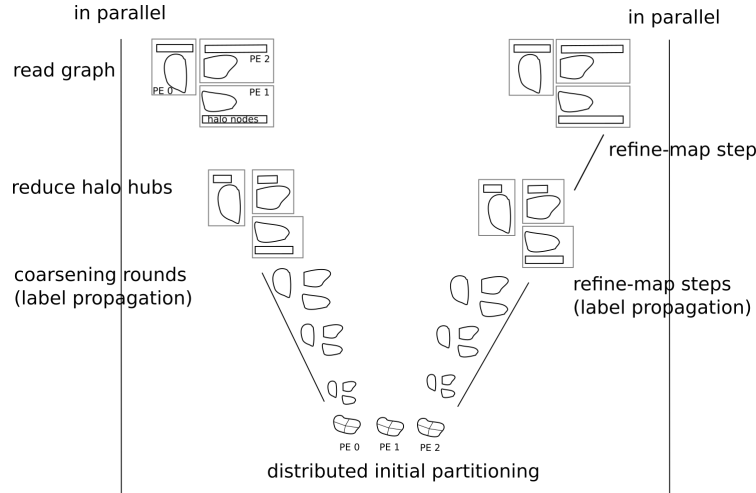
Fig. 3: Schematic interpretation of the parallel mapping for an example with three PEs and a partition in four blocks. The algorithm performs one multi-level cycle with additional pre- and post-processing steps to avoid memory issues.

PARHIP_MAP on its reduced graph and after completion each PE re-introduces the removed edges. Once the one multilevel cycle is complete, we perform a few extra rounds of parallel label propagation on the original graph to compensate for quality losses due to the re-introduction of the removed edges. This way we avoid the high memory consumption and save communication during coarsening and initial partition steps without sacrificing much of the quality. A schematic interpretation of the overall parallel algorithm is given in Figure 3.

4   Experiments

We perform experiments to evaluate the behavior of PARHIP_MAP on several graphs (see Table 1) downloaded from SNAP [22] or generated via KaGen [11] and ParMAT [18]. For disconnected graphs (in practice only some R-mat graphs) we extract the largest connected component. We implement PARHIP_MAP in C++, using the PARHIP graph API. For performance experiments, we compare against MPI-based partitioning tools, PARMETIS 4.0.3, PARHIP 3.10 (vanilla version configured to fastsocial) and XTRAPULP 0.3. As mentioned in Sec. 2.1, there are no direct competitors for MPI-based mapping solutions so we use partitioning with identity mapping. Often, identity mapping yields surprisingly good solutions, since it benefits from spatial locality [12]. Experiments were conducted on our local cluster[8] or the HLRN[9] cluster, Lise, in Berlin. Our local cluster contains 16 Linux machines, each equipped with an Intel Xeon X7460

---

[8] `https://www2.hu-berlin.de/macsy/technical-overview.html`
[9] `https://www.hlrn.de/`

CPU (2 sockets, 12 cores each), and 192 GB RAM. In Lise, each compute node has two Intel Cascade Lake Platinum 9242 CPUs with 384 GB RAM and 96 cores. Unless otherwise specified, we use one MPI process per compute node. We use the default settings for the competing algorithms, and similar build settings among all codes.[10] For all experiments, we report geometric mean results relative to ParHIP_map over all graphs (of Table 1). For each graph we repeat the experiment three times and we set the imbalance tolerance to 3%, one of the values used in [36] and in ParMETIS. To ensure reproducibility, all experiments were managed by SimexPal [2]. Our code and the experimental pipeline can be found at `https://github.com/hu-macsy/KaHIP`.

Table 1: 16 (undirected) graphs used in our experiments. Columns correspond to: name, type, number of vertices, number of edges, degree (average and max).

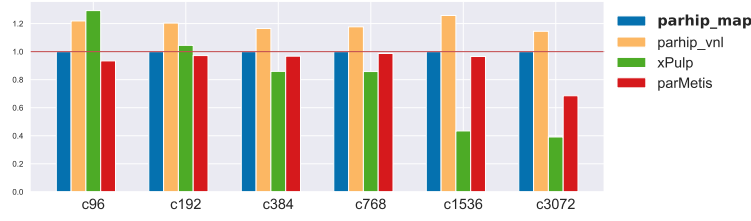| Network	Type	$|V|$	$|E|$	d_{avg}	d_{max}
coPapersCiteseer	REAL	434 102	16 036 720	73,8	1 188
eu-2005		862 664	16 138 468	37,4	68 963
as-skitter		1 696 415	11 095 298	13	35 455
orkut		3 072 441	117 184 899	76,2	33 313
dbpedia		18 265 512	136 535 446	14,9	612 308
friendster		65 608 366	1 806 067 135	55	612 308
twitter		52 515 193	1 963 197 641	74,7	3 691 240
r-mat (×3)	RMAT	$2^{22} - 2^{24}$	$2^{27} - 2^{29}$	40	18 484-63 345
barabasi-albert (×3)	BA	$2^{23} - 2^{27}$	$2^{26} - 2^{32}$	32	19 905-40 509
random-hyperbolic (×3)	RHG	$2^{25} - 2^{29}$	$2^{29} - 2^{33}$	16	83 645-200 165

4.1 Parallel Performance

We first evaluate the scalability behavior of ParHIP_map in a massively parallel setting of up to 3 072 PEs on Lise. In Figure 4a we report running times for all parallel tools relative to ParHIP_map. The results indicate that ParHIP_map exhibits a good scaling behavior. Compared to the fastest tools *i. e.*, xtraPulp and ParMETIS, ParHIP_map is on average only 18% and 9% slower, respectively. The high speed of xtraPulp is not surprising since it is designed to explicitly favor speed over quality. It is important to note that Figure 4a depicts aggregated results that may hide out-of-memory or timeout issues. After examining the failing rates, we observed that ParHIP_map has the smallest failing rate (17%), followed by xtraPulp, ParHIP and ParMETIS with failing rates of 42%, 62% and 65% respectively. To reflect a fairer comparison, we also include scalability experiments on our local cluster[11], where we observe a slightly better
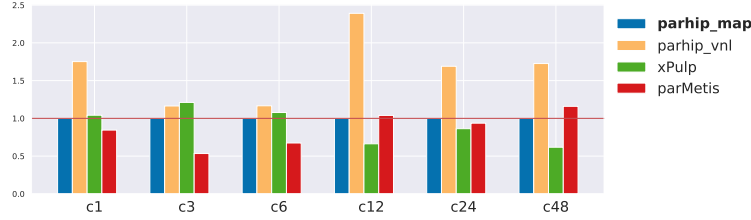
---

[10] On our local cluster: g++ 8.3.1 compiler with -O2 flags and the mpich 3.2 MPI library. On Lise: g++ 9.2 compiler with -O2 flags and openmpi 3.1.5.

[11] Here, we use one MPI process per core.

scaling behavior for ParHIP_map and similar trends for the other competitors (see Figure 4b). It is noteworthy to report that ParHIP_map is the only tool that runs successfully for the twitter graph. Precisely, our algorithm maps the twitter graph (a graph in the billion-scale range) into 384 blocks on 48 PEs of our local cluster in less than 6 minutes. ParMETIS and ParHIP failed for almost all Barabasi-Albert and R-mat graphs, probably due to the highly irregular degree distribution of these graphs, leading to memory issues. Moreover, in Figure 4c we report scaling results for an increasing number of blocks. We clearly see that ParHIP_map is on average 2× faster than ParHIP, slightly faster than ParMETIS and only about 0.7× slower than xtraPulp.

(a) Results for scaling PEs and constant number of blocks= 1536 on Lise.

(b) Results for scaling PEs and constant number of blocks= 384 on our local cluster.

(c) Results for scaling number of blocks and constant number of PEs= 24 on our local cluster.

Fig. 4: Relative running times for different scaling experiments on various clusters.

We also perform weak scaling experiments on Lise, for R-mat and random hyperbolic graphs of different sizes, for up to 768 and 1 536 PEs, respectively. For the experiment, we double the number of vertices as PEs double. The number of blocks is equal to the number of PEs used in the run and missing inputs are due to failing runs. In Figure 5, we see that PARHIP_MAP has a similar scaling behavior as XTRAPULP while the latter is faster as already observed from strong scaling.
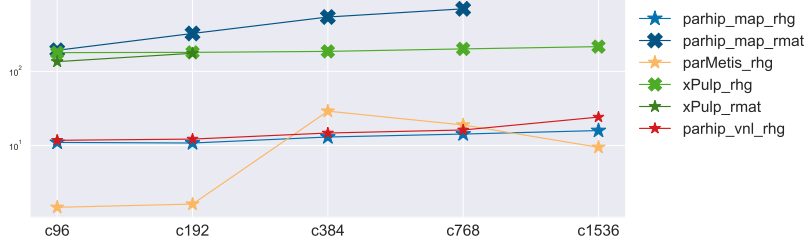

Fig. 5:   Absolute running times for weak scaling experiments on Lise (logarithmic scale).

4.2   Quality Results

To evaluate the solution quality, we use the objectives Coco and edgecut and run all parallel competitors as well as a sequential mapping approach from KAHIP, named here KAHIP_MAP[12], known to produce mapping results of high quality [10]. In Figure 6, we present relative Coco results for a constant number of PEs and an increasing number of blocks, since the latter often affects quality. For edgecut, we report results directly in the text, due to space limitations. Figure 6 shows that PARHIP_MAP achieves consistently the best Coco results compared to all other parallel approaches. More precisely, we are, on average, at least 4× better than XTRAPULP, 62% better than PARHIP, and 70% better than PARMETIS. Regarding edgecut, we are only 10% worse than the best competitor (PARHIP), 5% than PARMETIS, but 2.5× better than XTRAPULP. Those results are surprisingly good for our algorithm – given the fact that we do not optimize for edgecut, as the competitors do. Regarding the sequential baseline, KAHIP_MAP, we even achieve better quality (PARHIP_MAP is 10% better) and we are 30× faster using 24 PEs (as to 1 for KAHIP_MAP). Note that KAHIP_MAP, also fails to finish in time or has memory issues in many cases. Finally we should report that all tools occasionally fail to adhere to the balance constraint of 3% but do not largely overpass it either.

---

[12] Here, we set KAHIP_MAP to the fastsocial configuration.
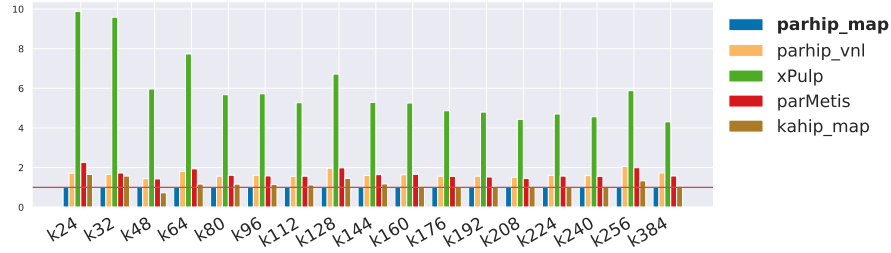
Fig. 6: Relative Coco results for 24 PEs (for the parallel tools). Lower is better.

5   Conclusions

In this work we propose a fully parallel mapping algorithm for distributed-memory systems. Our algorithm is an integrated solution *i. e.*, it addresses the partitioning and mapping problems simultaneously. We target hierachical systems and encode the hierarchy with a concise representation using bit-labels. Our approach exploits the above representation and uses parallel label propagation to devise a fast refinement process. As far as we know, this is the first parallel mapping algorithm for distributed-memory systems with a publicly available implementation (within ParHIP). Given the experimental results, our algorithm offers the best trade-off between mapping quality and speed compared to other MPI-based approaches. For future work we would like to integrate more scalable initial partitioning techniques (like the one proposed in [10]) to improve the performance of our current implementation.

References

1. Aktulga, H.M., Yang, C., Ng, E.G., Maris, P., Vary, J.P.: Topology-aware mappings for large-scale eigenvalue problems. In: Euro-Par 2012 Parallel Processing. pp. 830–842. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
2. Angriman, E., van der Grinten, A., von Looz, M., Meyerhenke, H., Nöllenburg, M., Predari, M., Tzovas, C.: Guidelines for experimental algorithmics: A case study in network analysis. Algorithms **12**(7), 127 (2019)
3. Barabási, A.L., Albert, R.: Emergence of scaling in random networks. Science **286**(5439), 509–512 (1999)
4. Bhatelé, A., Kalé, L.V., Kumar, S.: Dynamic topology aware load balancing algorithms for molecular dynamics applications. In: Proceedings of the 23rd International Conference on Supercomputing. p. 110–116. ICS '09, Association for Computing Machinery, New York, NY, USA (2009)

5. Bhatelé, A., Gupta, G.R., Kalé, L.V., Chung, I.: Automated mapping of regular communication graphs on mesh interconnects. In: 2010 International Conference on High Performance Computing. pp. 1–10 (2010)
6. Brandfass, B., Alrutz, T., Gerhold, T.: Rank Reordering for MPI Communication Optimization. Computers & Fluids **80**(0), 372 – 380 (2013). https://doi.org/http://dx.doi.org/10.1016/j.compfluid.2012.01.019
7. Buluç, A., Meyerhenke, H., Safro, I., Sanders, P., Schulz, C.: Recent advances in graph partitioning. In: Algorithm Engineering - Selected Results and Surveys, Lecture Notes in Computer Science, vol. 9220, pp. 117–158 (2016)
8. Chan, S.Y., Ling, T.C., Aubanel, E.: The Impact of Heterogeneous Multi-Core Clusters on Graph Partitioning: An Empirical Study. Cluster Computing **15**(3), 281–302 (2012)
9. Chen, H., Chen, W., Huang, J., Robert, B., Kuhn, H.: Mpipp: An automatic profile-guided parallel process placement toolset for smp clusters and multiclusters. In: Proceedings of the 20th Annual International Conference on Supercomputing. p. 353–360. ICS '06, Association for Computing Machinery, New York, NY, USA
10. Faraj, M.F., van der Grinten, A., Meyerhenke, H., Träff, J.L., Schulz, C.: High-Quality Hierarchical Process Mapping. In: 18th International Symposium on Experimental Algorithms (SEA 2020). vol. 160, pp. 4:1–4:15. Dagstuhl, Germany (2020)
11. Funke, D., Lamm, S., Sanders, P., Schulz, C., Strash, D., von Looz, M.: Communication-free massively distributed graph generation. In: 2018 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2018, Vancouver, BC, Canada, May 21 – May 25, 2018 (2018)
12. Glantz, R., Meyerhenke, H., Noe, A.: Algorithms for mapping parallel processes onto grid and torus architectures. In: 23rd Euromicro International Conference on Parallel, Distributed and Network-Based Processing, PDP 2015, Turku, Finland. pp. 236–243 (2015)
13. Glantz, R., Predari, M., Meyerhenke, H.: Topology-induced enhancement of mappings. CoRR **abs/1804.07131** (2018), `http://arxiv.org/abs/1804.07131`
14. Hoefler, T., Snir, M.: Generic Topology Mapping Strategies for Large-scale Parallel Architectures. In: ACM International Conference on Supercomputing (ICS'11). pp. 75–85. ACM (2011)
15. Hoefler, T., Jeannot, E., Mercier, G.: An Overview of Process Mapping Techniques and Algorithms in High-Performance Computing. In: High Performance Computing on Complex Environments, pp. 75–94. Wiley (Jun 2014)
16. Jeannot, E., Mercier, G., Tessier, F.: Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. IEEE Transactions on Parallel and Distributed Systems **PP**(99), 1–1 (2013). https://doi.org/10.1109/TPDS.2013.104
17. Jeannot, E., Mercier, G., Tessier, F.: Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. IEEE Transactions on Parallel and Distributed Systems **25**(4), 993 – 1002 (2014)
18. Khorasani, F., Gupta, R., Bhuyan, L.N.: Scalable simd-efficient graph processing on gpus. In: Proceedings of the 24th International Conference on Parallel Architectures and Compilation Techniques. pp. 39–50. PACT '15 (2015)
19. Kirchbach, K.V., Schulz, C., Träff, J.L.: Better process mapping and sparse quadratic assignment. ACM J. Exp. Algorithmics **25** (Sep 2020)
20. Kirmani, S., Park, J., Raghavan, P.: An embedded sectioning scheme for multiprocessor topology-aware mapping of irregular applications. IJHPCA **31**(1), 91–103 (2017)

21. Kleinberg, J.: The small-world phenomenon: An algorithmic perspective. In: Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing. p. 163–170. STOC '00, Association for Computing Machinery, New York, NY, USA (2000)
22. Leskovec, J.: Stanford Network Analysis Package (SNAP), `http://snap.stanford.edu/index.html`
23. Low, Y., Bickson, D., Gonzalez, J., Guestrin, C., Kyrola, A., Hellerstein, J.M.: Distributed graphlab: A framework for machine learning and data mining in the cloud. Proc. VLDB Endow. **5**(8), 716–727 (Apr 2012)
24. Meyerhenke, H., Sanders, P., Schulz, C.: Parallel graph partitioning for complex networks. IEEE Trans. Parallel Distributed Syst. **28**(9), 2625–2638 (2017)
25. Pellegrini, F.: Static Mapping by Dual Recursive Bipartitioning of Process and Architecture Graphs. In: Scalable High-Performance Computing Conference (SHPCC). pp. 486–493. IEEE (May 1994)
26. Pellegrini, F.: Scotch and libScotch 5.0 User's Guide. Tech. rep., LaBRI, Université Bordeaux I (December 2007)
27. Pellegrini, F.: Static Mapping of Process Graphs. In: Graph Partitioning, chap. 5, pp. 115–136. John Wiley & Sons (2011)
28. Pellegrini, F.: Scotch and PT-Scotch Graph Partitioning Software: An Overview. In: Naumann, U., Schenk, O. (eds.) Combinatorial Scientific Computing, pp. 373–406. CRC Press (2012)
29. Raghavan, U.N., Albert, R., Kumara, S.: Near linear time algorithm to detect community structures in large-scale networks. Physical Review E **76**(3) (2007)
30. Salihoglu, S., Widom, J.: Gps: A graph processing system. In: Scientific and Statistical Database Management. Stanford InfoLab (July 2013)
31. Sanders, P., Schulz, C.: Distributed evolutionary graph partitioning. In: Proceedings of the Meeting on Algorithm Engineering & Expermiments. p. 16–29. ALENEX '12, Society for Industrial and Applied Mathematics, USA (2012)
32. Sanders, P., Schulz, C.: Kahip v0.53 - karlsruhe high quality partitioning - user guide. CoRR **abs/1311.1714** (2013)
33. Schloegel, K., Karypis, G., Kumar, V.: Parallel Static and Dynamic Multi-Constraint Graph Partitioning. Concurrency and Computation: Practice and Experience **14**(3), 219–240 (2002)
34. Slota, G.M., Rajamanickam, S., Devine, K., Madduri, K.: Partitioning trillion-edge graphs in minutes. In: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS). pp. 646–655 (2017)
35. Soo-Young Lee, Aggarwal: A mapping strategy for parallel processing. IEEE Transactions on Computers **C-36**(4), 433–442 (1987)
36. Walshaw, C., Cross, M.: Jostle: Parallel multilevel graph-partitioning software – an overview. In: Magoules, F. (ed.) Mesh Partitioning Techniques and Domain Decomposition Techniques, pp. 27–58. Civil-Comp Ltd. (2007), (Invited chapter)