Brief Announcement: Distributed Unconstrained Local Search for Multilevel Graph Partitioning

Peter Sanders
sanders@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

Daniel Seemaier
daniel.seemaier@kit.edu
Karlsruhe Institute of Technology
Karlsruhe, Germany

ABSTRACT

Partitioning a graph into blocks of *roughly equal* weight while cutting only few edges is a fundamental problem in computer science with numerous practical applications. While shared-memory parallel partitioners have recently matured to achieve the same quality as widely used sequential partitioners, there is still a pronounced quality gap between distributed partitioners and their sequential counterparts. In this work, we shrink this gap considerably by describing the engineering of an unconstrained local search algorithm suitable for distributed partitioners. We integrate the proposed algorithm in a distributed multilevel partitioner. Our extensive experiments show that the resulting algorithm scales to thousands of PEs while computing cuts that are, on average, only 3.5% larger than those of a state-of-the-art high-quality shared-memory partitioner. Compared to previous distributed partitioners, we obtain on average 6.8% smaller cuts than the best-performing competitor while being more than 9 times faster.

KEYWORDS

graph partitioning, distributed algorithms, parallel algorithms

1 INTRODUCTION

Graphs are a central concept of computer science used whenever we need to model relations between objects. Consequently, handling *large* graphs is very important for parallel processing. This often requires to *partition* these graphs into blocks of roughly equal weight while cutting only few edges between the blocks (*balanced* graph partitioning).

More precisely, consider a graph $G = (V = 1..n, E, c, \omega)$ with positive vertex weights c and edge weights ω. We are looking for a partition Π of V into k non-overlapping blocks $V_1, \ldots, V_k$. The *balance constraint* demands that for all blocks $c(V_i) \leq L_{\max} := (1 + \varepsilon)\frac{c(V)}{k}$ for some imbalance parameter ε. The objective is to minimize the *cut*, i.e., the total weight of all inter-block edges.

There has been a huge amount of research on graph partitioning, so we refer the reader to overview papers [1] for most of the general material. *Multilevel graph partitioners* (e.g., [5, 6, 8]) achieve high-quality partitions for a wide range of input graphs with a good trade-off between quality and partitioning cost. First, iteratively *coarsen* the graph G. The resulting small graph G' is a good representation of the input and an *initial partition* of G' already induces a good partition of G. This is further improved by *uncoarsening* the graph and improving the partition on each level through refinement algorithms. Arguably the most successful refinement algorithm is the *Fiduccia-Mattheyses* local search algorithm [3] (FM).

However, parallelizing the refinement phase has proved challenging over several decades. Shared-memory partitioners have recently matured to achieve high quality and reasonable scalability [5] through an efficient parallelization of the FM algorithm, requiring fine-grained synchronization and communication, which is unsuitable for distributed memory. On the other hand, distributed partitioners [6, 8] rely on simpler local search algorithms such as label propagation, inducing a considerable quality penalty. Label propagation visits each vertex in parallel and moves the vertex to the block that minimizes the induced cut while preserving the balance constraint. This process is usually repeated for multiple rounds and terminated once no more vertices are moved during a round (i.e., the partition is in a local optimum) or a maximum number of rounds has been exceeded.

Recently, Gilbert et al. [4] introduced the refinement algorithm *Jet*, which they show to achieve similar partition quality to FM refinement while being suitable for GPU parallelism. The Jet algorithm first builds a set of *move candidates* M: for each vertex v in block V_i, let $\text{conn}(v, V')$ be the summed edge weight between v and vertices in block V'. Let $V_j = \arg\max_{V'} \text{conn}(v, V')$ be the block with the strongest connection to v. Then, v is included in M if $\text{conn}(v, V_j) - \text{conn}(v, V_i) \geq -\lfloor \tau \cdot \text{conn}(v, V_i) \rfloor$, where τ is a constant *temperature* parameter. For $\tau = 1$, all vertices are included, whereas $\tau = 0$ only considers vertices with positive gain. Note that in contrast to *standard* label propagation, M might also include moves that violate the balance constraint. The moves in M are then sorted by their gain value before positive gain moves are applied, locking them in the next round to avoid oscillation. If violated, an independent *rebalancing* algorithm fixes the balance constraint afterward.

Based on the idea of balance-violating moves with subsequent rebalancing, Maas et al. [7] developed an *unconstrained FM* adaptation as part of the shared-memory parallel partitioner Mt-KaHyPar. Their experimental evaluation shows that unconstrained FM achieves considerable improvements compared to regular FM (and also Jet) on real-world graphs with a power-law degree distribution.

Contributions. We achieve similar quality and much better scalability by porting Jet to a distributed-memory setting. Move generation is relatively easy to adapt but further improved by working with multiple temperatures. We develop an alternative highly scalable probabilistic algorithm for rebalancing, that can perform a large number of favorable balancing moves in parallel. We finish up using the slower but more controlled algorithm from dKaMinPar [9]. It remains to be seen whether this approach is more suitable if k is large. Extensive experiments on real-world graphs and huge randomly generated networks indicate scalability to (at least) 8 192 cores and one trillion (directed) edges. Compared to state-of-the-art distributed partitioners we achieve considerably improved quality when partitioning into $k \in \{2, 4, \dots, 128\}$ blocks.

2 DISTRIBUTED JET REFINEMENT

We consider a graph G distributed across P processing elements (PEs) $1..P$, each containing a subgraph of consecutive vertices of G distributed s.t. each PE has roughly the same *number of edges*.

Undirected edges $\{u, v\}$ are represented by two directed edges (u, v) and (v, u), which are stored on the PEs owning the respective tail vertices. Vertices adjacent to vertices owned by other PEs are called *interface vertices* and replicated as *ghost vertices* (i.e., without outgoing edges). The partition $\Pi = \{V_1, \dots, V_k\}$ is distributed s.t. each PE stores the block IDs for its vertices, including its ghost vertices.

Jet Refinement. Implementing Jet [4] in this model requires only minor modifications. We restate the algorithm here in order to be self-contained. First, recall that each PE builds a set of local move candidates M by visiting its owned vertices, including a vertex v of block V_i in M if $g(v) \coloneqq \max_{j \neq i} \mathrm{conn}(v, V_j) - \mathrm{conn}(v, V_i) \geq -\lfloor \tau \cdot \mathrm{conn}(v, V_i) \rfloor$. The temperature τ controls the extent of move candidates with negative gain (i.e., that would increase the partition cut) included in M. Note that $\mathrm{conn}(v, \cdot)$ can be computed without communication since it only depends on v's neighborhood. Next, each interface vertex $v \in M$ sends its corresponding $g(v)$ value to its ghost replicates. With this information, PEs independently re-evaluate their move candidates, removing all vertices v that would increase the partition cut assuming that any neighbor u with $g(u) > g(v)$ is moved before v. The remaining vertices are moved to their target blocks, locked for the next round, and the block assignment of ghost replicates is updated accordingly.

Rebalancing. Rebalancing the partition afterwards while retaining most of its improved quality is crucial to the overall performance of the algorithm. Ref. [9] introduces a distributed rebalancing algorithm that can only move a few vertices per overloaded block in every centrally coordinated epoch. This induces a sequential bottleneck for partitions with highly overloaded blocks. To make rebalancing scalable, we extend the algorithm by a round of the following highly parallel approach whenever a single round reduces the total partition overload by less than 10%.

Following Ref. [9], we consider vertices $v \in V_o$ of overloaded blocks $V_o \in \Pi$ and look at the maximum reduction in edge cut g_v when moving v to any non-overloaded block. Then, each PE sorts its vertices into exponentially spaces buckets b_o^j based on their *maximum relative gain*, defined as $r_v \coloneqq g_v \cdot c(v)$ if $g_v > 0$ and $r_v \coloneqq g_v / c(v)$ otherwise. A vertex v with $r_v \geq 0$ is assigned to bucket $j = 0$, whereas the bucket of negative gain vertices is calculated as $j = 1 + \lceil \log_\alpha (1 - r_v) \rceil$. During early experimentation, we observed that the choice of α considerably impacts the resulting partition quality after rebalancing. Thus, we use $\alpha = 1.1$ to obtain our results. After constructing the PE-local buckets, we compute global bucket weights $c(B_o^i) \coloneqq \sum_p c(b_o^i @ p)$ using an all-reduce operation, where $b_o^i @ p$ refers to the value of variable b_o^i on PE p. The *cut-off buckets* $\hat{B}_o \coloneqq \min \left\{ j \mid \sum_{i<j} c(B_o^i) \geq c(V_o) - L_{\max} \right\}$ contain the lowest-rated vertices that must be moved to remove all excess weight from overloaded blocks. Since moving all vertices in buckets up to the cut-off buckets could introduce overloaded target blocks, we use a probabilistic approach that maintains balance (of non-overloaded blocks) in expectation.[1] To this end, let W_u be the total vertex weight of all move candidates in buckets up to the cut-off bucket with target block V_u. Then, move vertices to V_u with probability $p_u \coloneqq (L_{\max} - c(V_u)) / W_u$ (if $p_u \geq 1$, we always perform the move).

Integration. We perform $t = 4$ rounds of the proposed algorithm. Since we have observed that Jet is very sensitive to the temperature used in the construction of M, we perform multiple rounds of the algorithm to achieve more robust performance. During the i-th round, we set the temperature to $\tau_i = \tau_0 + \frac{i}{t}(\tau_1 - \tau_0)$, where $\tau_0 = 0.75$ and $\tau_1 = 0.25$. Following Ref. [4], a single round consists of repeating the described Jet refinement and rebalancing steps until 12 consecutive repetitions did not improve the partition.

3 EXPERIMENTS

We have integrated the proposed refinement algorithm into the distributed multilevel partitioner dKaMinPar [9], compiled all codes using g++-12.1 with flags `-O3 -march=native` and use OpenMPI 4.0 as parallelization library. We use the benchmark set of Ref. [9] (32 graphs) to evaluate the solution quality of the proposed refinement algorithm, partitioning each graph using $k \in \{2, 4, \dots, 128\}$ and $\varepsilon = 3\%$ on a single machine equipped with a 64-core AMD EPYC 7702P and 1 TB of main memory.

Looking at Fig. 1a, we observe that d4xJet improves the partition cut by at least 10% on roughly 50% of all instances compared to plain dKaMinPar, which only uses label propagation for refinement. This is considered a lot in the context of multilevel graph partitioning, as most multilevel partitioners achieve average edge cuts within a few percentage points of each other.

Compared to the state-of-the-art shared-memory partitioner Mt-KaHyPar [7], we find that d4xJet computes partitions with at most 1% larger cuts on 50% of all instances, shrinking the quality gap

[1] Note that a more straightforward adaptation of the rebalancing algorithm proposed by Ref. [4] to distributed memory could work as follows. As proposed in Ref. [4], only consider moves to target blocks with weights below the *dead zone* $L_{\max} - \delta \cdot (L_{\max} - c(V)/k)$, where $\delta \in [0, 1]$ is a tuning parameter. Then, for each overloaded block V_o, always apply all moves in buckets below the cut-off bucket $\hat{B}_o$. Additionally, let PE p move the highest rated vertices of the cut-off bucket $\hat{B}_o$ with total weight proportional to the total weight of block V_o on PE p. While this process could introduce new overloaded blocks, it terminates in $\leq k$ iterations as at least one block per round is moved into the dead zone. It is open to see which approach gives better performance, especially for large k.

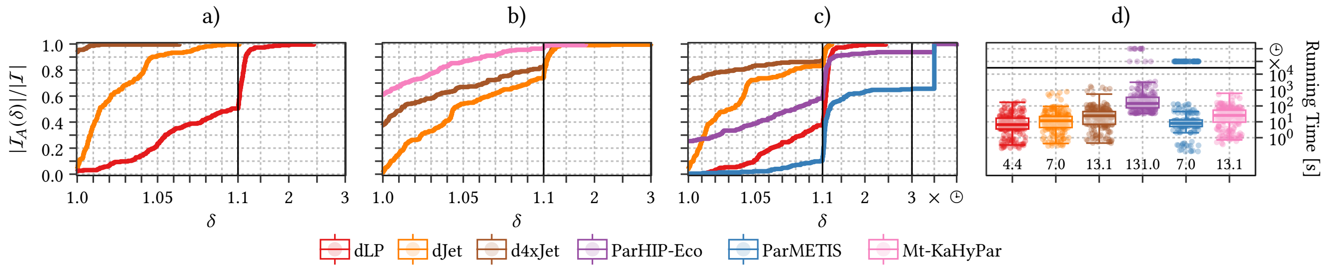

Figure 1: Solution quality comparison using performance profiles. Let $\mathcal{A}$ be the set of algorithms, $\mathcal{I}$ the set of instances, and $q_A(I)$ the edge cut of algorithm $A \in \mathcal{A}$ on instance $I \in \mathcal{I}$. For each algorithm A, we plot the fraction of instances $\frac{|\mathcal{I}_A(\delta)|}{|\mathcal{I}|}$ (y-axis) where $\mathcal{I}_A(\delta) := \{I \in \mathcal{I} \mid q_A \leq \delta \cdot \min_{A' \in \mathcal{A}} q_{A'}(I)\}$ and δ on the x-axis. (a) Solution quality of plain dKaMinPar (dLP) and with 1 resp. 4 rounds of Jet refinement (dJet resp. d4xJet). (b) Comparison against Mt-KaHyPar [7] (unconstrained FM refinement). (c) Comparison against distributed partitioners ParHIP [8] and ParMETIS [6]. (d) Per-instance running times with gmeans across all instances that were solved by all algorithms or exceeded the time limit (1 h).

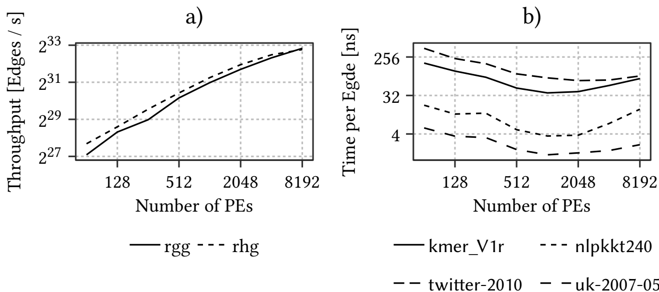

Figure 2: Results for d4xJet. (a) Weak scaling on random geometric (rgg) and hyperbolic (rhg, power-law exponent 3.0) graphs with 2^{21} vertices and 2^{27} (directed) edges per core. (b) Strong scaling on real-world graphs with low (kmer_V1r, nlpkkt240) and high (twitter-2010, uk-2007-05) max. degree.

between distributed and shared-memory partitioners considerably (Fig. 1b). On average, the cuts found by Mt-KaHyPar are 3.5% smaller than those of d4xJet, with roughly equal running times (Fig. 1d).

Fig. 1c compares against competing distributed partitioners. d4xJet finds at least ≈ 8% lower cuts on roughly 50% of all instances than ParHIP-Eco, which obtained the best cuts in Ref. [9]. More, ParHIP-Eco is considerably slower than d4xJet due to its expensive evolutionary initial partitioning algorithm.

To evaluate the scalability of d4xJet, we perform additional weak- and strong-scaling experiments on $\{1, 2, \ldots, 128\}$ nodes of the HoreKa high-performance cluster, setting $k = 16$ and $\varepsilon = 3\%$. Each node is equipped with two 38-core Intel Xeon Platinum 8368 processors[2]. The nodes are connected by an InfiniBand 4X HDR 200 Gbit/s network with approx. 1 μs latency. Results are shown in Fig. 2, where we observe weak scalability up to (at least) 8 192 cores on randomly generated graphs. On medium-sized real-world graphs, we generally observe scalability up to 1 024–2 048 cores. This is roughly in line with competing distributed partitioners (see, e.g., [9]), which we discuss briefly. ParMETIS is unable to partition the graphs with high maximum degree due to its matching-based coarsening strategy, does not scale on kmer_V1r, but scales up to 2 048 cores on nlpkkt240 with speedup 8.7 (over 64 cores; d4xJet scales to 1 024 cores with speedup 5.3). ParHIP(-Fast) fails to partition the twitter-2010 and kmer_V1r graphs regardless of the number of cores used and achieves a speedup of 2.1 on 2 048 cores for uk-2007-05 (d4xJet achieves 3.9), and 4.5 on 512 cores on nlpkkt240. Interestingly, we note that edge cuts improve slightly when increasing the number of cores.

[2]We only use 64 out of the available 76 cores because some of the graph generators require the number of cores to be a power of two.

4 CONCLUSION

We have engineered a distributed implementation of the Jet algorithm [4], shrinking the quality gap between distributed and shared-memory partitioners considerably while scaling to thousands of cores. Further improvements might be possible by using more sophisticated local search algorithms. For instance, Pt-Scotch [2] uses the FM algorithm on the *band graph* induced by the partition (local copies of the subgraph surrounding the cut), but this imposes a sequential bottleneck on graphs with large cuts. Another direction for future work could be a Jet-based partitioner for distributed GPUs.

ACKNOWLEDGMENTS

This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 882500). This work was performed on the HoreKa supercomputer funded by the Ministry of Science, Research and the Arts Baden-Württemberg and by the Federal Ministry of Education and Research.

REFERENCES

[1] Ü. V. Çatalyürek et al. 2022. More Recent Advances in (Hyper)Graph Partitioning. *ACM Comput. Surv.* (2022).
[2] C. Chevalier and F. Pellegrini. 2008. PT-Scotch: A Tool for Efficient Parallel Graph Ordering. *Parallel Comput.* 34, 6–8 (2008), 318–331.
[3] C. M. Fiduccia and R. M. Mattheyses. 1982. A linear-time heuristic for improving network partitions. In *19th Design Automation Conf., (DAC)*. ACM/IEEE, 175–181.
[4] M. S. Gilbert, K. Madduri, E. G. Boman, and S. Rajamanickam. 2023. Jet: Multilevel Graph Partitioning on GPUs. *CoRR* abs/2304.13194 (2023). arXiv:2304.13194
[5] L. Gottesbüren, T. Heuer, P. Sanders, and S. Schlag. 2021. Scalable Shared-Memory Hypergraph Partitioning. In *ALENEX 2021*. SIAM, 16–30.
[6] G. Karypis and V. Kumar. 1998. Multilevel k-way Partitioning Scheme for Irregular Graphs. *J. Parallel Distributed Comput.* 48, 1 (1998), 96–129.
[7] Nikolai Maas, Lars Gottesbüren, and Daniel Seemaier. 2024. Parallel Unconstrained Local Search for Partitioning Irregular Graphs. In *ALENEX 2024*. SIAM, 32–45.
[8] H. Meyerhenke, P. Sanders, and C. Schulz. 2017. Parallel Graph Partitioning for Complex Networks. *IEEE Trans. Parallel Distrib. Syst.* 28, 9 (2017), 2625–2638.
[9] P. Sanders and D. Seemaier. 2023. Distributed Deep Multilevel Graph Partitioning. In *Euro-Par 2023: Parallel Processing*. Springer Nature Switzerland, 443–457.