Engineering Graph Partitioning Algorithms

Vitaly Osipov, Peter Sanders, Christian Schulz

Karlsruhe Institute of Technology (KIT), 76128 Karlsruhe, Germany
{osipov, sanders, christian.schulz}@kit.edu

Abstract. The paper gives an overview of our recent work on balanced graph partitioning – partition the nodes of a graph into k blocks such that all blocks have approximately equal size and such that the number of cut edges is small. This problem has numerous applications for example in parallel processing. We report on a scalable parallelization and a number of improvements on the classical multi-level approach which leads to improved partitioning quality. This includes an integration of flow methods, improved local search, several improved coarsening schemes, repeated runs similar to the approaches used in multigrid solvers, and an integration into a distributed evolutionary algorithm. Overall this leads to a system that for many common benchmarks leads to both the best quality solution known and favorable tradeoffs between running time and solution quality.

1 Introduction

Graph partitioning is a common technique in computer science, engineering, and related fields. For example, good partitionings of unstructured and irregular graphs are very valuable in the area of *high performance computing*, e.g., when solving *partial differential equations*. These equations are usually discretized and then solved numerically using a parallel computer, e.g. using a CG method. To effectively balance the load we need a graph model of computation and communication. Roughly speaking, vertices in the graph represent computation units and edges denote communication. Now this graph needs to be partitioned such that there are few edges between the blocks (pieces). In particular, when we want to solve the partial differential equation in parallel on k PEs (processing elements) we want to partition the graph into k blocks of about equal size. In this paper we focus on a version of the problem that constrains the maximum block size to $(1 + \epsilon)$ times the average block size and tries to minimize the total cut size, i.e., the number of edges that run between blocks.

A successful heuristics for partitioning large graphs is the *multilevel* approach depicted in Figure 1 where the graph is recursively *contracted* to achieve a smaller graph with the same basic structure. After applying an *initial partitioning* algorithm to this small graph, the contraction is undone and, at each level, a *local refinement* method is used to improve the partitioning induced by the coarser level. Refer to [1–4] for overviews on existing methods. Although several successful multilevel partitioners have been developed in the last 14 years, we had the impression that certain aspects of the method are not well understood. We therefore have built our own graph partitioner KaPPa [5] (Karlsruhe Parallel Partitioner) with focus on scalable parallelization. Somewhat astonishingly, we also obtained improved partitioning quality through rather simple methods. This motivated us to make a fresh start putting all aspects of MGP on

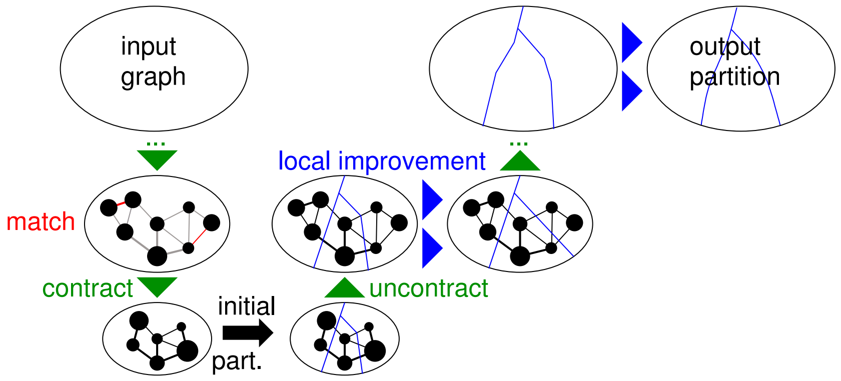

Fig. 1. Multilevel graph partitioning.

trial. This paper gives an overview of our recent work on balanced graph partitioning. We present four algorithms: KaPPa [5], KaSPar [6] (Karlsruhe Sequential Partitioner) which contracts only a single edge per level, KaFFPa [7] (Karlsruhe Fast Flow Partitioner) which uses advanced refinement techniques, and the distributed evolutionary algorithm, KaFFPa(E)volutionary [8]. We only give a short outline of the main ideas and refer to the respective papers for more details.

To give some minimalistic experimental data, we consider Walshaw's benchmark [9] which consists of 34 graphs with up to 3.3 million edges, $k \in \{2, 4, 8, 16, 32, 64\}$ blocks and imbalance $\epsilon \in \{0, 1\%, 3\%, 5\%\}$. Excluding the case $\epsilon = 0$ which our codes do not handle yet, we obtain 612 instances. For each algorithm we will report the number of instances where we are the record holder or at least as good as the record holder. On the first glance it looks more fair to exclude ties but this would complicate the figure since we would also have to differentiate between ties with our own codes and improvements that entered the archive recently. Moreover, for $k = 2$ and many of the smaller graphs no improvements have been found for a long time which indicates that the solutions might already be optimal.

2 *Ka*rlsruhe *P*arallel *Pa*rtitioner [5]

We now present our parallel approach to graph partitioning. First of all the graph is distributed among all k PEs. This is done by computing a preliminary partition of the graph. Currently we have implemented a recursive bisection algorithm for nodes with 2D coordinates that alternately splits the data by the x-coordinate and the y-coordinate [10, 11]. We can also use the initial numbering of the nodes.

Now we have to compute matchings to create coarser versions of the graph. Following the basic approach from [12] we combine a sequential matching algorithm running on each PE and a parallel matching algorithm running on the *gap graph*. The gap graph consists of those edges $\{u, v\}$ where u and v reside on different PEs and $\omega(\{u, v\})$

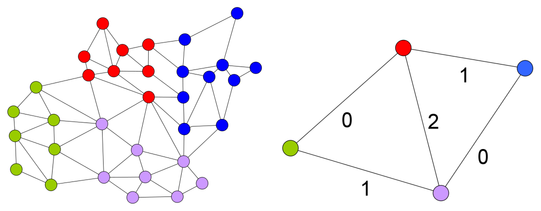

Fig. 2. A graph which is partitioned into four blocks and its corresponding quotient graph $\mathcal{Q}$. The quotient graph has an edge coloring indicated by the numbers and each edge set induced by edges with the same color form a matching $\mathcal{M}(c)$. Pairs of blocks with the same color can be refined in parallel.

exceeds the weight of the edges that may have been matched by the local matching algorithms to u and v.

In [5], expanding on an idea already present in [13], we proposed to make contraction more systematic by separating two issues: A *rating function* indicates how much sense it makes to contract an edge based on *local* information. A *matching* algorithm tries to maximize the sum of the ratings of the contracted edges looking at the *global* structure of the graph. While the rating functions allows us a flexible characterization of what a "good" contracted graph is, the simple, standard definition of the matching problem allows us to reuse previously developed algorithms for weighted matching. Matchings are contracted until the graph is "small enough". In most previous work, the edge weight $\omega(e)$ itself is used as a rating function (see [5] for more details). We have shown in [5] that the rating function $\mathrm{expansion}^{*2}(\{u,v\}) := \frac{\omega(\{u,v\})^2}{c(u)c(v)}$ works best among other edge rating functions where $c(\cdot)$ is the node weight – usually the number of input nodes contracted into a node in the current graph.

We employed the *Global Path Algorithm (GPA)* as sequential matching algorithm. It was proposed in [14] as a synthesis of the Greedy algorithm and the Path Growing Algorithm [15]. This algorithm achieves a half-approximation in the worst case, but empirically, GPA gives considerably better results than Sorted Heavy Edge Matching and Greedy (for more details look into [5]). Our implementation of the parallel matching algorithm proposed in [12] iteratively matches edges in the gap graph $\{u, v\}$ that are locally heaviest both at u and v until no more edges can be matched.

The contraction is stopped when the number of remaining nodes is small enough. We employ Scotch [16] as an initial partitioner since it empirically performs better than Metis [3]. This algorithm is then run simultaneously on all PEs, each with a different seed for the random number generator. The best solution is then broadcast to all PEs.

Recall that the refinement phase iteratively uncontracts the matchings contracted during the contraction phase. After a matching is uncontracted, local search based refinement algorithms move nodes between block boundaries in order to reduce the cut while maintaining the balancing constraint. As most other current systems, we adopt the FM-algorithm [17] which runs in linear time. The main difference of our approach to previous systems is that at any time, each PE may work on only one pair of neighboring blocks performing a local search constrained to moving nodes between these two blocks. Thus, we need parallel algorithms for deciding which processors work on which pairs of blocks. For this purpose, we use the *quotient graph* Q whose nodes are blocks of the current partition and whose edges indicate that there are edges between these blocks in the underlying graph G. Since we have the same number of PEs and blocks, each PE will work on the block assigned to it and at one of its neighbors in Q. Figure 2 gives an example. We schedule all possible pairwise refinement operations by using a distributed parallel edge coloring algorithm on the quotient graph. The edges in each color class define a matching in the quotient graph and can be handled in parallel. The PEs at both endpoints of an edge work on the same pairwise refinement operation redundantly but using different random number seeds in tie breaking operations. After the local search is finished, the better partitioning of the two blocks is adopted.

In the past, parallelizing a graph partitioner meant giving up some quality for speed. Somewhat surprisingly, this was not the case for KaPPa. In the Walshaw benchmark we obtain 189 best values, in particular for the largest graphs. Besides the improvements like edge ratings that are also easy to integrate into previous solvers, one source of improvement was the more focused local search. Our interpretation is that this makes it more likely that the local search is successful.

3 The n-Level approach [6]

The success of focusing local search in KaPPa gave us the idea to drive this focusing idea into the extreme. This fits well the the idea of an n-level multilevel algorithm that we have previously used successfully for route planning [18] and the nearest neighbor problem [19].

The central idea behind this (sequential) approach we called KaSPar (Karlsruhe Sequential Partitioner) [20] is to make subsequent levels as similar as possible – we (un)contract only a *single* edge between two levels. We call this n-GP since we have (almost) n levels of hierarchy. Figure 1 gives a high-level recursive summary of n-GP. We use similar edge rating functions and initial partitioning as in KaPPa. However, note that no matching algorithm is needed. Rather, on each level, we choose a single edge to be contracted using a priority queue.

In order to make contraction and uncontraction efficient, we use a "semidynamic" graph data structure: When contracting an edge $\{u, v\}$, we mark both u and v as deleted, introduce a new node w, and redirect the edges incident to u and v to w. The advantage of this implementation is that edges adjacent to a node are still stored in adjacency arrays which are more efficient than linked lists needed for a full fledged dynamic graph data structure. A disadvantages of our approach is a certain space overhead. However, it is relatively easy to show that this space overhead is bounded by a logarithmic factor even if we contract edges in some random fashion (see [21]). Overall, with respect to asymptotic memory overhead, n-GP is no worse than methods with a logarithmic number of levels.

The local search strategy is similar to the FM-algorithm [17]. We now outline our variant. Initially, all nodes are unmarked and inactive. The neighbors of the recently expanded edge are activated. Active nodes reside in priority queues – one for each block

Algorithm 1 n-GP(G, k, ϵ).

```
if $G$ is small then
    return initialPartition$(G, k, \epsilon)$
pick the edge $e = \{u, v\}$ with highest rating
contract $e$; $\mathcal{P} := n{-}GP(G, k, \epsilon)$; uncontract $e$
activate$(u)$;   activate$(v)$;   localSearch()
return $\mathcal{P}$
```

they could be moved to. The key for the priority queue is the *gain*, i.e., the decrease in edge cut when the node is moved (which can also be negative). We call a queue P_B eligible if the highest gain node in P_B can be moved to block B without violating the balance constraint for block B. Local search repeatedly looks for the highest gain node v in any eligible priority queue P_B and moves v to block B. When this happens, node v becomes nonactive and marked, the unmarked neighbors of v get activated and the gains of the active neighbors are updated. The local search is stopped if either no eligible nonempty queues remain or some additional stopping criterion hold. After the local search stopped, it is rolled back to the lowest cut state reached during the search. Subsequently all previously marked nodes are unmarked. The local search is repeated until no improvement is achieved.

The problem with this approach is that the local search would usually spread over the whole graph which in conjunction with the linear number of levels could lead to quadratic running time. We therefore introduced additional stopping criteria. First of all, our local search does nothing if none of the uncontracted nodes is a *border node*, i.e., has a neighbor in another block. Other FM-algorithms initialize the search with all border nodes. Indeed, our experiments indicate that for large graphs and small number of partitions k, the overall local search effort may grow sublinearly with the input size. To achieve this we also need a way to abort unpromising local searches that *are* started. We do this by modelling the local search as a random walk on the cut size axis. If within this model it becomes unlikely to reach an overall improvement in a linear number of steps, we abort the search.

KaSPar achieves 238 best values for the Walshaw benchmark although it is a much simpler code than KaPPa.

4 ***Ka*rlsruhe *F*ast *F*low *P*artitioner [7]**

In order to implement further ideas for improving the quality of sequential multilevel graph partitioning we went back to the traditional variant with a logarithmic number of levels – this allows us to use simple and fast static graph representations and enables a more global view on refinement. We still exploit the advantages of highly focused local search by performing many focused searches initialized with a single border node.

Max-Flow Min-Cut Local Improvement. A more global method is based on max-flow min-cut computations between pairs of blocks, in other words, a method to improve a given bipartition. Roughly speaking, this improvement method is applied between all pairs of blocks that share a nonempty boundary. The algorithm basically constructs a flow problem by growing an area around the given boundary vertices of a pair of blocks such that each s-t cut in this area yields a feasible bipartition of the original graph/pair of blocks *within* the balance constraint. One can then apply a max-flow min-cut algorithm to obtain a min-cut in this area and therefore an improved cut between the original pair of blocks. This can be improved in multiple ways, for example, by iteratively applying the method, searching in larger areas for feasible cuts, and applying most balanced minimum cut heuristics.

Global Search. KaFFPa extends the concept of *iterated multilevel algorithms* which was introduced by [22] and can be traced back to multigrid solvers for sparse systems of linear equations. The main idea is to iterate the coarsening and uncoarsening phase. Once the graph is partitioned, edges that are between two blocks are not contracted. An F-cycle works as follows: on *each* level we perform at most *two recursive calls* using different random seeds during contraction and local search. A second recursive call is only made the second time that the algorithm reaches a particular level. Figure 3 illustrates a F-cycle. As soon as the graph is partitioned, edges that are between blocks are not contracted. This ensures nondecreasing quality of the partition since our refinement algorithms guarantee no worsening and break ties randomly. These so called *global search strategies* are more effective than plain restarts of the algorithm.

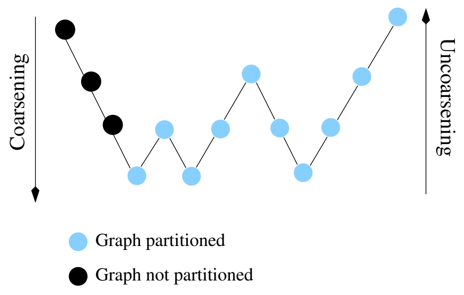

Fig. 3. An F-cycle.

KaFFPa achieves 435 best values in the Walshaw benchmark.

5 KaFFPa*E*volutionary [8]

In the Walshaw benchmark, KaFFPa was beaten mostly for small graphs that combine multilevel partitioning with an evolutionary algorithm. We therefore developed an improved evolutionary algorithm that also employs coarse grained parallelism. Roughly speaking, KaFFPaE uses KaFFPa to create individuals and modifies the coarsening phase to provide new effective combine operations. We restrict ourselves to the description of the combine operator framework and the parallelization.

An EA starts with a population of individuals (in our case partitions of the graph) and evolves the population into different populations over several rounds. In each round, the EA uses a selection rule based on the fitness of the individuals (in our case the edge cut) of the population to select good individuals and combine them to obtain improved offspring.

Combine Operators. In KaFFPaE we have a general combine operator framework, i.e. a partition $\mathcal{P}$ can be combined with another partition of the population or an arbitrary clustering of the graph. This is achieved by running a modified version of KaFFPa that during coarsening will not contract edges that are cut in one of the input partitions/clusterings. As soon as the coarsening phase is stopped, we apply the partition $\mathcal{P}$ to the coarsest graph and use this as initial partitioning. This way the resulting partition is at least as good as the input partition and in addition, the refinement algorithms can effectively exchange good parts of the solution on the coarse levels by moving only a few vertices.

Parallelization. We parallelize the evolutionary algorithm on p processing elements (PEs), by distributing the population over the PEs. Basically, every PE runs a sequential evolutionary algorithm on its subpopulation. To achieve fast initialization for large p, each PE will begin with a single individual. Subsequently, using random cyclic communication patterns, each PE is equipped with a random selection of these initial individuals. After initialization, some interaction is achieved by periodically sending the best local solution to a random PE. Communication volume is limited by sending the same solution at most $\log p$ times. This has been implemented using MPI.

KaFFPaE achieves 470 best values in the Walshaw benchmark, investing two hours of time on the small to medium sized graphs and eight hours of time on the eight largest graphs of the archive, on two somewhat outdated 8-core nodes. It should also be noted that very good values are already achieved in seconds to minutes of parallel execution time so that the solutions can also be used for many applications rather than only for record hunting. The parallelization scales well to hundreds of processors even for small graphs.

6 10th DIMACS Implementation Challenge

We coorganized a DIMACS implementation challenge on graph partitioning and clustering[1]. An important outcome is collection of benchmark graphs that not only has more instances than the Walshaw collection but also more varied applications and larger graphs with up to 3 billion edges. Our partitioners also work very well on these instances achieving the best marks both with respect to quality and running time versus quality among all participants. A surprising result was obtained for a part of the challenge where the objective function was not cut size but a measure of communication volume. This objective function can be expressed as a hypergraph partitioning problem. Interestingly, KaFFPaE outperformed dedicated hypergraph partitioners by just changing the fitness function to prefer solutions with low communication volume – the multilevel

[1] `http://www.cc.gatech.edu/dimacs10/`

algorithm still optimizes cuts. This is an indication that some of our techniques would also be useful in a multilevel algorithm for optimizing communication volume or even for a general hypergraph partitioner.

7 Conclusions and Future Work

The perspective taken in this paper is that we developed our graph partitioners KaPPa, KaSPar, KaFFPa, and KaFFPaE in a benchmark driven way achieving overall 550 out of 612 optimal entries in the Walshaw benchmark with $\epsilon > 0$. Another equally valid perspective is that we applied the methodology of algorithm engineering to all aspects of the multi-level graph partitioning approach, achieving improvements in coarsening, refinement, parallelization, global search guidance, and embedding into metaheuristics.

Both perspectives also allow an outlook on open problems: Within the Walshaw benchmark, the perfectly balanced case ($\epsilon = 0$) is an interesting case requiring new techniques to obtain good solutions fast. The DIMACS challenge indicates that for difficult instances like social networks, a lot of work remains to be done. Considerations for massively parallel computing including petascale and exascale indicate that we have to go back to scalable parallelization for big instances that do not fit into internal memory of a single node and that much larger values of k will become relevant. On the other hand, its also not clear how to employ many processors for small values of k. This is important for recursive partitioning schemes that are useful in many cases and can also more easily adapt to hierarchical computer architectures.

A feature oriented view has initial partitioning as an obvious open point where we still do not have our own solution – currently we are using Scotch [16] in all our systems. Another interesting issue are new approaches to coarsening. In [23] we are cooperating on an approach where nodes are fractionally assigned to nodes on the coarser levels. This gives us more flexibility in shaping the coarse levels and seems promising for social networks and other graphs that have problems with edge contraction approaches.

Yet another view could be the different activities in algorithm engineering. The reported work is strong on design, implementation, experimental evaluation and benchmarking but weak on other aspects. We plan to release an easy to use algorithm library with some of our codes soon. Theoretical analysis of complex metaheuristics like KaFFPaE is very difficult. However note that an astonishingly large set of "easier" graph algorithms are used "under the hood" that are theoretically better understood: weighted matching, spanning trees, edge coloring, BFS, shortest paths, diffusion, maximum flows, and strongly connected components. Perhaps this is one justification why a group coming from algorithm theory can be successful in real world graph partitioning. Yet the most interesting activity from algorithm engineering for finding future work is modeling. Applications indicate that we should also look at other objective functions (e.g., communication volume, separator size, and bottleneck variants), hypergraph partitioning, and clustering (where balance is not directly relevant and k may not be known or flexible). Some of our techniques like edge ratings, F-cycles, or n-level contraction might also be relevant for other multilevel graph algorithms, e.g., for graph drawing.

References

1. Fjallstrom, P.: Algorithms for graph partitioning: A survey. Linkoping Electronic Articles in Computer and Information Science **3**(10) (1998)
2. G. Karypis, V.K.: A fast and high quality multilevel scheme for partitioning irregular graphs. SIAM JOURNAL ON SCIENTIFIC COMPUTING **20**(1) (1998) 359–392
3. Schloegel, K., Karypis, G., Kumar, V.: Graph partitioning for high performance scientific simulations. Technical Report 00-018, University of Minnesota (2000)
4. Walshaw, C., Cross, M.: JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In Magoules, F., ed.: Mesh Partitioning Techniques and Domain Decomposition Techniques. Civil-Comp Ltd. (2007) 27–58 (Invited chapter).
5. Holtgrewe, M., Sanders, P., Schulz, C.: Engineering a Scalable High Quality Graph Partitioner. 24th IEEE International Parallal and Distributed Processing Symposium (2010)
6. Osipov, V., Sanders, P.: n-Level Graph Partitioning. 18th European Symposium on Algorithms (see also arxiv preprint arXiv:1004.4024) (2010)
7. Sanders, P., Schulz, C.: Engineering Multilevel Graph Partitioning Algorithms. 19th European Symposium on Algorithms (2011)
8. Sanders, P., Schulz, C.: Distributed Evolutionary Graph Partitioning. 12th Workshop on Algorithm Engineering and Experimentation (2011)
9. Soper, A., Walshaw, C., Cross, M.: A combined evolutionary search and multilevel optimisation approach to graph-partitioning. Journal of Global Optimization **29**(2) (2004) 225–241
10. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM **18**(9) (1975) 509–517
11. Berger, M.J., Bokhari, S.H.: A partitioning strategy for pdes across multiprocessors. In: ICPP. (1985) 166–170
12. Manne, F., Bisseling, R.H.: A parallel approximation algorithm for the weighted maximum matching problem. In: 7th Int. Conf. on Parallel Processing and Applied Mathematics (PPAM). Volume 4967 of LNCS., Springer (2007) 708–717
13. Abou-Rjeili, A., Karypis, G.: Multilevel algorithms for partitioning power-law graphs. In: International Parallel & Distributed Processing Symposium. (2006)
14. Maue, J., Sanders, P.: Engineering algorithms for approximate weighted matching. In: 6th Workshop on Exp. Algorithms (WEA). Volume 4525 of LNCS., Springer (2007) 242–255
15. Drake, D., Hougardy, S.: A simple approximation algorithm for the weighted matching problem. Information Processing Letters **85** (2003) 211–213
16. Pellegrini, F.: Scotch home page. `http://www.labri.fr/pelegrin/scotch`
17. Fiduccia, C.M., Mattheyses, R.M.: A Linear-Time Heuristic for Improving Network Partitions. In: 19th Conference on Design Automation. (1982) 175–181
18. Geisberger, R., Sanders, P., Schultes, D., Vetter, C.: Exact routing in large road networks using contraction hierarchies. Transportation Science (2012) to appear.
19. Birn, M., Holtgrewe, M., Sanders, P., Singler, J.: Simple and fast nearest neighbor search. In: 11th Workshop on Algorithm Engineering and Experiments (ALENEX). (2010)
20. Osipov, V., Sanders, P.: n-level graph partitioning. In: 18th European Symposium on Algorithms (ESA). Volume 6346 of LNCS. (2010) 278–289
21. Dementiev, R., Sanders, P., Schultes, D., Sibeyn, J.: Engineering an external memory minimum spanning tree algorithm. In: IFIP TCS, Toulouse (2004) 195–208
22. Walshaw, C.: Multilevel refinement for combinatorial optimisation problems. Annals of Operations Research **131**(1) (2004) 325–372
23. Safro, I., Sanders, P., Schulz, C.: Advanced coarsening schemes for graph partitioning. In: 11th International Symposium on Experimental Algorithms. LNCS (2012)