Engineering State-of-the-Art Graph Partitioning Libraries @KIT

Vitaly Osipov, Peter Sanders, Christian Schulz, Manuel Holtgrewe
Karlsruhe Institute of Technology, *Karlsruhe, Germany*
Email: {osipov, sanders, christian.schulz}@kit.edu, manuel.holtgrewe@fu-berlin.de

Abstract

We describe two different approaches to multi-level graph partitioning (MGP). The first is an approach to parallel graph partitioning that scales to hundreds of processors. All components of this algorithm are implemented by scalable parallel algorithms. The second algorithm is based on the extreme idea to contract only a single edge on each level of the hierarchy. This obviates the need for a matching algorithm and promises very good partitioning quality since there are very few changes between two levels. Both algorithms produce a very high solution quality. Quality improvements compared to previous systems are due to better prioritization of edges to be contracted, better approximation algorithms for identifying matchings and FM local search algorithms that work more locally than previous approaches.

1 Introduction

Graph partitioning is a common technique in computer science, engineering, and related fields. Good partitionings of unstructured and irregular graphs are very valuable in the area of *high performance computing*. An important example are *partial differential equations*. These equations are usually 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.[1]

A successful heuristic 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 induces by the coarser level. In this paper we present two algorithms: a scalable approach to parallel graph partitioning and the n-Level approach to graph partitioning. Somewhat astonishingly, we indeed found several opportunities for improvement with significant impact on partitioning quality and scalability.

After introducing basic concepts in Section 2, we present both algorithms in Section 3. This is followed by the evaluation of both algorithms and comparisons with other state-of-the-art graph partitioners in Section 4. Related work can be found in Section 5.

2 Preliminaries

Consider an undirected graph $G = (V, E, c, \omega)$ with edge weights $\omega : E \to \mathbb{R}_{>0}$, node weights $c : V \to \mathbb{R}_{\geq 0}$, $n = |V|$, and $m = |E|$. We extend c and ω to sets, i.e., $c(V') := \sum_{v \in V'} c(v)$ and $\omega(E') := \sum_{e \in E'} \omega(e)$. $\Gamma(v) := \{u : \{v, u\} \in E\}$ denotes the neighbors of v.

[1] It is well known that there are more realistic (and more complicated) objective functions involving also the block that is worst and the number of its neighboring nodes [14] but minimizing the cut size has been adopted as a kind of standard since it is usually highly correlated with the other formulations. We believe that the results presented here will be adaptable to other objective functions and also to other setting such as graph clustering where k and the block sizes are not necessarily fixed.

We are looking for *blocks* of nodes $V_1,\ldots,V_k$ that partition V, i.e., $V_1 \cup \cdots \cup V_k = V$ and $V_i \cap V_j = \emptyset$ for $i \neq j$. The *balancing constraint* demands that $\forall i \in 1..k : c(V_i) \leq L_{\max} := (1+\epsilon)c(V)/k + \max_{v \in V} c(v)$ for some parameter ϵ. The last term in this equation arises because each node is atomic and therefore a deviation of the heaviest node has to be allowed. The objective is to minimize the total *cut* $\sum_{i<j} w(E_{ij})$ where $E_{ij} := \{\{u,v\} \in E : u \in V_i, v \in V_j\}$. By default, our initial inputs will have unit edge and node weights. However, even those will be translated into weighted problems in the course of the algorithm.

A matching $M \subseteq E$ is a set of edges that do not share any common nodes, i.e., the graph (V, M) has maximum degree one. An edge coloring $\mathcal{C}$ assigns a color (a number) to each edge of a graph such that no two incident edges have the same color. Note that the edges with a particular color define a matching, i.e., $\mathcal{C}$ partitions the edges into matchings. We will be interested in colorings with a small number of different colors used.

Contracting an edge $\{u,v\}$ means to replace the nodes u and v by a new node x connected to the former neighbors of u and v. We set $c(x) = c(u) + c(v)$ so the weight of a node at each level is the number of nodes it is representing in the original graph. If replacing edges of the form $\{u,w\}, \{v,w\}$ would generate two parallel edges $\{x,w\}$, we insert a single edge with $\omega(\{x,w\}) = \omega(\{u,w\}) + \omega(\{v,w\})$.

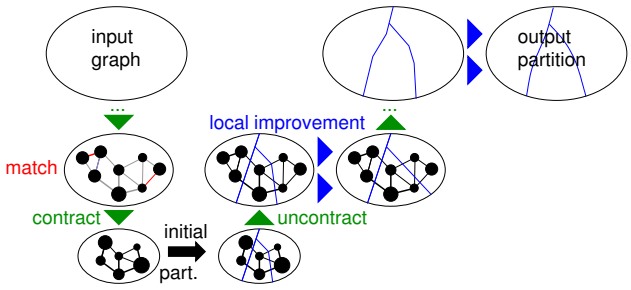

Figure 1: Multilevel graph partitioning.

Uncontracting an edge e undos its contraction. In order to avoid tedious notation, G will denote the current state of the graph before and after a (un)contraction unless we explicitly want to refer to different states of the graph.

The multilevel approach to graph partitioning consists of three main phases. In the *contraction* (coarsening) phase, we iteratively identify matchings $M \subseteq E$ and contract the edges in M. This is repeated until $|V|$ falls below some threshold. Contraction should quickly reduce the size of the input and each computed level should reflect the global structure of the input network. In particular, nodes should represent densely connected subgraphs.

Contraction is stopped when the graph is small enough to be directly partitioned in the *initial partitioning phase* using some other algorithm. We could use a trivial initial partitioning algorithm if we contract until exactly k nodes are left. However, if $|V| \gg k$ we can afford to run some expensive algorithm for initial partitioning.

In the *refinement* (or uncoarsening) phase, the matchings are iteratively uncontracted. After uncontracting a matching, the refinement algorithm moves nodes between blocks in order to improve the cut size or balance. The nodes to move are often found using some kind of local search. The intuition behind this approach is that a good partition at one level of the hierarchy will also be a good partition on the next finer level so that refinement will quickly find a good solution.

3 Two approaches to graph partitioning

3.1 The parallel approach

We now present our approach to parallel graph partitioning. First of all the graph is distributed among all k PEs. This is done by computing a preliminary partition of the graph, e.g., using coordinate information. 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 [1, 2]. We can also use the initial numbering of the nodes.

Now we have to compute matchings to create coarser versions of the graph. 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 $\omega(\{u,v\})$ exceeds the weight of the edges that may have been matched by the local matching algorithms to u and v.

In [15] 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 (see also Section 5). 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 Section 5 for more details). In [15] we have shown that the rating function $\text{expansion}^{*2}(\{u,v\}) := \frac{\omega(\{u,v\})^2}{c(u)c(v)}$ works best among other edge rating functions.

We employed the *Global Path Algorithm (GPA)* as sequential matching algorithm. It was proposed in [18] as a synthesis of the Greedy algorithm and the Path Growing Algorithm [7]. 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 [15]). The GPA scans the edges in order of decreasing weight but rather than immediately building a matching, it first constructs a collection of paths and even cycles. Afterwards, optimal solutions are computed for each of these paths and cycles using dynamic programming. Our implementation of the parallel matching algorithm proposed in [17] iteratively matches edges $\{u,v\}$ 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 on some PE is below $\max(20, n/(\alpha k^2))$ for some tuning parameter α. The graph is then small enough to be partitioned on a single PE. We employ Scotch as an initial partitioner since it empirically performs better than Metis. 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.

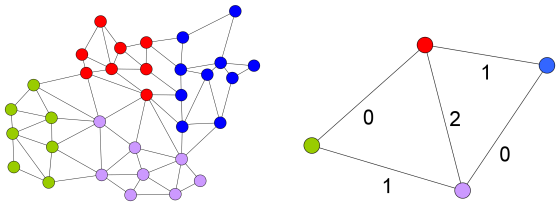

Figure 2: A graph which is partitioned into four blocks and its corresponding quotient graph $\mathcal{Q}$. The quotient graph has an edge coloring indicated by the numbers and each edge set induced by edges with the same color form a matching $\mathcal{M}(c)$. Pairs of blocks with the same color can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

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 basic approach from [8] 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. From now on, we will therefore identify blocks and PEs. Figure 2 gives an example.

We use matchings of Q to define with which neighbor in Q a PE is working at a particular point in time. If $\{u,v\}$ is in the matching, both corresponding PEs will refine the partitions u and v using different seeds for their random number generator. See below for more details. After the local search is finished, the better partitioning of the two blocks is adopted. We employ the following strategy to find pairs of blocks for refinement: we step through the colors of an edge coloring of the quotient graph Q. Our coloring algorithm is a parallelization of a well known sequential greedy edge coloring algorithm. It can be shown that this algorithm needs at most twice as many colors as an optimal edge coloring. For more details look into [15, 27]. We now describe the refinement between two blocks. Before a local search operation, we perform a bounded breadth first search starting from the boundary of each block, and send copies of this boundary array to the partner PE in the local search. The local search is then limited to this boundary area. This way, for large graphs, only a small fraction of each block has to be communicated. If it should really happen that the local search would profit from going beyond the boundary area, this will be possible in the next iteration of some of the outer loops. Figure 3 shows this schematically.

The local search algorithm itself is basically the FM-algorithm [8]: For each of the two blocks A, B under consid-

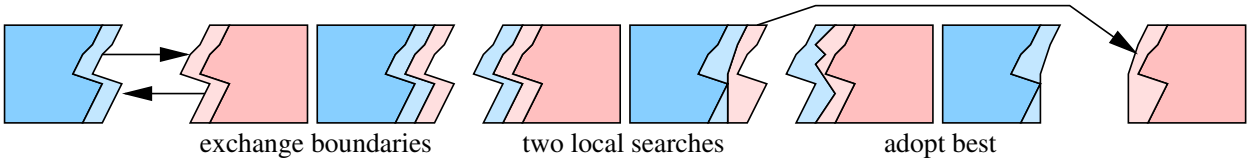

Figure 3: Refinement between two blocks using boundary exchange.

eration, a PE keeps a priority queue of nodes eligible to move. The priority is based on the *gain*, i.e., the decrease in edge cut when the node is moved to the other side. Each node is moved at most once within a single local search. The queues are initialized in random order with the nodes at the partition boundary. We have tried several queue selection strategies: *Alternating* between A and B [8], *MaxLoad* where always the heavier block gives a node, and *TopGain*, where the queue promising larger gain is used. While MaxLoad yields partitions which are very balanced, TopGain produces partitions with higher quality within the balance constraint.

The search is broken when more than $\alpha \min\{|A|, |B|\}$ nodes have been moved without yielding an improvement. When the search stops, search is rolled back to the state with the lexicographically best value of the tuple $(imbalance, cutValue)$. Where $imbalance$ is $\max(0, \max(c(A) - L_{\max}, c(B) - L_{\max}))$.

3.2 The n-Level approach

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

Figure 4: n-GP.

We now describe our approach to sequential graph partitioning. Here our central idea is to get even better partitions by making 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 4 gives a high-level recursive summary of n-GP. Again we use Scotch as a base case partitioner when the graph is sufficiently small. In KaSPar, contraction is stopped when either only $20k$ nodes remain, no further nodes are eligible for contraction, or there are less edges than nodes left. The latter happens when the graph consists of many independent components. The edges to be contracted are chosen according to an edge rating function. KaSPar also adopts the rating function $\text{expansion}^{*2}(\{u, v\})$. Additionally, in order to avoid unbalanced inputs to the initial partitioner, KaSPar never allows a node v to participate in a contraction if the weight of v exceeds $1.5n/(20k)$. Selecting contracted edges can be implemented efficiently by keeping the contractable *nodes* in a priority queue sorted by the rating of their most highly rated incident edge.

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 [6]). 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 [8]. We now outline our variant and then discuss differences. Originally, all nodes are unmarked. Only unmarked nodes are allowed to be activated or moved from one block to another. Activating a node $v \in B'$ means that for blocks $\{B \neq B' : \exists \{v, u\} \in E \wedge u \in B\}$ we compute the gain

$$g_B(v) = \sum \{\omega(\{v, u\}) : \{v, u\} \in E, v \in B\} - \sum \{\omega(\{v, u\}) : \{v, u\} \in E, v \in B'\}$$

of moving v to block B for blocks where v can be moved. Note that gains are allowed to be negative. Node v is then inserted into the priority queue P_B using $g_B(v)$ as the priority. 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 one of the stopping criteria described below applies. After the local search stops, it is rolled back to the lowest cut state reached during the search (which is the starting state if no improvement was achieved). Subsequently all previously marked nodes are unmarked. The local search is repeated until no improvement is achieved.

The main difference to the usual FM-algorithm is that our routine performs a highly localized search starting just at the uncontracted edge. Indeed, 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. In n-GP the local search may find an improvement quickly after moving a small number of nodes. However, in order to exploit this case, we need a way to stop the search much earlier than previous algorithms which limit the number of steps to a constant fraction of the current number of nodes $|V|$.

Stopping Using a Random Walk Model. It makes sense to make a stopping rule more adaptive by making it dependent on the past history of the search, e.g., on the difference between the current cut and the best cut achieved before.

We model the gain values in each step as identically distributed, independent random variables whose expectation μ and Variance σ^2 is obtained from the previously observed p steps. In [21] we show how from these assumptions we can (heuristically) derive that it is unlikely that the local search will produce a better cut if

$$p\mu^2 > \alpha\sigma^2 + \beta \tag{1}$$

where α and β are tuning parameters and μ is the average gain since the last improvement. For the variance σ^2, we can also use the variance observed throughout the current local search. Parameter β is a base value that avoids stopping just after a small constant number of steps that happen to have small variance. Currently we set it to $\ln n$.

4 Experiments

Implementation We have implemented the algorithm described above using C++ and MPI. Hash tables use the library (extended STL) provided with the GCC compiler. For the following comparisons we used Scotch 5.1, Metis 4.0 and ParMetis 3.1.1.

System We have run our code on a cluster with 200 nodes each equipped with two Quad-core Intel Xeon processors (X5355) which run at a clock speed of 2.667 GHz, have 2x4 MB of level 2 cache each and run Suse Linux Enterprise 10 SP 1. All nodes are attached to an InfiniBand 4X DDR interconnect which is characterized by its very low latency of below 2 microseconds and a point to point bandwidth between two nodes of more than 1300 MB/s. Our program was compiled using GCC Version 4.3.1 and optimization level 3 using OpenMPI 1.2.8. Henceforth, a PE is one core of this machine.

Instances We report experiments on two suites of instances summarized in Table 1. $rggX$ is a *random geometric graph* with 2^X nodes where nodes represent random points in the unit square and edges connect nodes whose Euclidean distance is below $0.55\sqrt{\ln n/n}$. This threshold was choosen in order to ensure that the graph is almost connected. $DelaunayX$ is the Delaunay triangulation of 2^X random points in the unit square. Graphs $fetooth..auto$ come from Chris Walshaw's benchmark archive [29]. Graphs deu and eur are undirected versions of the road networks of Germany, and Western Europe respectively, used in [5]. Instance $af\_shell10$ comes from the Florida Sparse Matrix Collection [4]. For the number of partitions k we choose the values used in [29]: 2, 4, 8, 16, 32, 64. Our default value for the allowed inbalance is 3 % since this one is the default value in Metis. When not otherwise mentioned, we

Large instances		
graph	n	m
rgg20	2^{20}	13 783 240
Delaunay20	2^{20}	12 582 744
fetooth	78 136	905 182
598a	110 971	1 483 868
ocean	143 437	819 186
144	144 649	2 148 786
wave	156 317	2 118 662
m14b	214 765	3 358 036
auto	448 695	6 629 222
deu	4 378 446	10 967 174
eur	18 029 721	44 435 372
af_shell10	1 508 065	51 164 260

algorithm	large graphs		
	best	avg.	t[s]
KaSPar strong	12 450	12 584	87.12
KaPPa strong	13 323	+8%	28.16
Scotch	14 475	+19%	3.83
kMetis	15 540	+32%	0.71

Table 1: On the left hand side: Basic properties of the graphs from our benchmark set. On the right hand side the geometric means (times, cut values) over all instances.

perform 10 repetitions of each run and report the average result. When averaging over multiple instances, we use the geometric mean in order to give every instance the same influence on the final figure. [2]

4.1 Comparison with other Partitioners

We now use our suite of larger graphs to compare our algorithm variants KaPPa Strong and KaSPar Strong (see [15, 21] for more details and configuration of the algorithms) with Scotch, Metis. Table 1 compares the performances of these algorithms. For kMetis the differences are 32 % for the strong variant of KaSPar and 22 % for the strong variant of KaPPa. For Scotch, we get 19 % and 10 % respectively. Note that these differences are much larger than what can be obtained by just repeated runs, which gives only about 3 % improvement for 10 repetitions. Comparing average execution times of parallel KaPPa with the sequential algorithms makes little sense because this depends a lot on the number of PEs used.

For the largest graphs available to us, we have scaled the number of processors further up to 1024. In Figure 5 we see that KaPPa[3] scales well all the way to the largest number of processors, while parMetis reaches its limit of scalability at around 100 PEs. Eventually, parMetis is slower than the fastest variant of KaPPa.

[2] Because we have multiple repetitions for each instance, we compute the geometric mean of the average (avg.) edge cut values for each instance or the geometric mean of the best (best.) edge cut value occurred.

[3] The minimal variant scales up to 512 PEs but this could be repaired by breaking the contraction later.

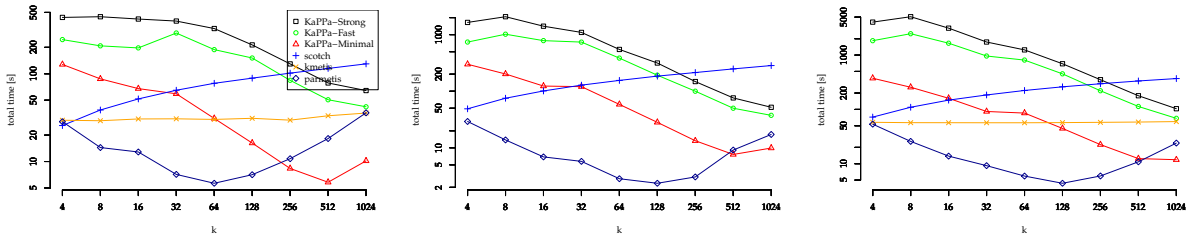

Figure 5: Scalability of KaPPa variants for graphs eur, rgg25, and delaunay25.

5 Related Work

This paper is a summary of [15] and [21]. There has been a huge amount of research on graph partitioning so that we refer to introductory and overview papers such as [9, 10, 26, 30] for a general overview. Well-known software packages based on MGP are Chaco [13], DiBaP [20], Jostle [31, 30], Metis [16], Party [24, 25], and Scotch [22, 23].

There is a long tradition of n-level algorithms in geometric data structures based on randomized incremental construction (e.g, [12, 3]). Our motivation for studying n-level are *contraction hierarchies* [11], a preprocessing technique for route planning that is at the same time simpler and an order of magnitude more efficient than previous techniques using a small number of levels.

The parallel version of Jostle [30] is similar to our approach since it applies local search to pairs of neighboring partitions. However, this parallelization has problems maintaining the balance of the partitions since at any particular time, it is difficult to say how many nodes are assigned to a particular block. We solve this problems by performing concurrent local searches only on independent pairs of partitions.

PT-Scotch, the parallel version of Scotch is based on recursive bipartitioning. This is more difficult to parallelize than direct k-partitioning since in the initial bipartition, there is less parallelism available. The unused processor power is used by performing several independent attempts in parallel. The involved communication effort is reduced by considering only nodes close to boundary of the current partitioning (band-refinement). We also use band-refinement but using a different algorithm and with much less replication of work.

DiBaP [19] is a multi-level graph partitioning package based on diffusion. It previously yielded the best partitioning results for the biggest graphs in [28] but has no scalable parallelization.

References

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[2] M. J. Berger and S. H. Bokhari. A partitioning strategy for pdes across multiprocessors. In *ICPP*, pages 166–170, 1985.

[3] M. Birn, M. Holtgrewe, P. Sanders, and J. Singler. Simple and fast nearest neighbor search. In *2010 Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, January*, volume 16, pages 43–54, 2010.

[4] T. Davis. The University of Florida Sparse Matrix Collection, `http://www.cise.ufl.edu/research/sparse/matrices`, 2008.

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

[6] R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *IFIP TCS*, Toulouse, 2004.

[7] D. Drake and S. Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85:211–213, 2003.

[8] C. M. Fiduccia and R. M. Mattheyses. A Linear-Time Heuristic for Improving Network Partitions. In *19th Conference on Design Automation*, pages 175–181, 1982.

[9] P.O. Fjallstrom. Algorithms for graph partitioning: A survey. *Linkoping Electronic Articles in Computer and Information Science*, 3(10), 1998.

[10] V. Kumar G. Karypis. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM JOURNAL ON SCIENTIFIC COMPUTING*, 20(1):359–392, 1998.

[11] R. Geisberger, P. Sanders, D. Schultes, and D. Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *7th Workshop on Experimental Algorithms (WEA)*, volume 5038 of *LNCS*, pages 319–333. Springer, 2008.

[12] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(4):381–413, 1992.

[13] B. Hendrickson. Chaco: Software for partitioning graphs. `http://www.sandia.gov/~bahendr/chaco.html`.

[14] B. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? (extended abstract). In *IRREGULAR*, pages 218–225, 1998.

[15] M. Holtgrewe, P. Sanders, and C. Schulz. Engineering a scalable high quality graph partitioner. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE, 2010.

[16] George Karypis and Vipin Kumar. MeTiS, A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices, Version 4.0, 1998.

[17] F. Manne and R. H. Bisseling. 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*, pages 708–717. Springer, 2007.

[18] J. Maue and P. Sanders. Engineering algorithms for approximate weighted matching. In *6th Workshop on Exp. Algorithms (WEA)*, volume 4525 of *LNCS*, pages 242–255. Springer, 2007.

[19] H. Meyerhenke, B. Monien, and T. Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions of very high quality. In *IEEE International Symposium on Parallel and Distributed Processing, 2008. IPDPS 2008.*, pages 1–13, 2008.

[20] Henning Meyerhenke, Burkhard Monien, and Thomas Sauerwald. A new diffusion-based multilevel algorithm for computing graph partitions. *Journal of Parallel and Distributed Computing*, 69(9):750–761, 2009.

[21] V. Osipov and P. Sanders. n-Level Graph Partitioning. *ESA 2010*, pages 278–289.

[22] Francois Pellegrini. SCOTCH: Static Mapping, Graph, Mesh and Hypergraph Partitioning, and Parallel and Sequential Sparse Matrix Ordering Package, 2007. `http://www.labri.fr/perso/pelegrin/scotch/`.

[23] Francois Pellegrini. SCOTCH 5.1 User's guide. Technical report, Laboratoire Bordelais de Recherche en Informatique, Bordeaux, France, 2008.

[24] Robert Preis. PARTY Partitioning Library , 1996. `http://www2.cs.uni-paderborn.de/cs/robsy/party.html`.

[25] Robert Preis and Ralf Diekmann. The PARTY Partitioning Library, User Guide . Technical report, University of Paderborn, Germany, 1996. Tr-rsfb-96-02.

[26] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. In J. Dongarra et al., editor, *CRPC Par. Comp. Handbook*. Morgan Kaufmann, 2000.

[27] C. Schulz. Scalable parallel refinement of graph partitions. 2009.

[28] C. Walshaw. The Graph Partitioning Archive, `http://staffweb.cms.gre.ac.uk/~c.walshaw/partition/`, 2008.

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

[30] C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. In F. Magoules, editor, *Mesh Partitioning Techniques and Domain Decomposition Techniques*, pages 27–58. Civil-Comp Ltd., 2007. (Invited chapter).

[31] Chris Walshaw. JOSTLE –graph partitioning software, 2005. `http://staffweb.cms.gre.ac.uk/~wc06/jostle/`.