Dissertationsschrift

Shape Optimized Graph Partitioning

zur Erlangung des Grades eines
Doktors der Naturwissenschaften

von

Stefan Schamberger

September 2006

Fakultät für Elektrotechnik, Informatik und Mathematik
Universität Paderborn

Contents

1 Introduction

The efficient usage of parallel computing resources plays a key role for many large scale scientific simulations. These applications usually follow the single-instruction multiple-data (SIMD) concept and consist of a huge number of small calculations that operate iteratively and are related via data dependencies. In order to minimize the overall computation time, an efficient parallelization requires these tasks to be distributed equally among all processing nodes.

Because of the dependencies between the calculations, a data distribution involves communication in the processing network. Hence, additional costs in terms of latency and bandwidth occur and have to be considered when determining the data assignment onto the processors.

The calculations and their dependencies can be modeled by the vertices and edges of a graph. The objective to split this graph into equally sized parts, such that the number of edges between different parts is minimal, is known as the *graph-partitioning* problem. It occurs as a subproblem in many applications and is difficult to solve. Though several analytical bounds for special cases of this problem are known, their proofs are either not constructive nor are the involved algorithms suited for practical use. Hence, heuristics have been developed that deliver good solutions quickly, and these are widely applied in the field of parallel and distributed computing.

The most successful heuristics are described in chapter 2. The *Spectral Bisection* is based on properties derived from the algebraic representation of the graph and assigns the vertices according to an eigenvector. The popular linear ordering mechanisms induced by *Space-Filling Curves* rely on geometric information. These are very fast, but they run into problems if the provided coordinates do not correlate with the graph structure. As an enhancement, we introduce the *Graph-Filling Curves*, a new linear ordering approach that reflects the connectivity information more closely and therefore enables better distributions. To improve a given partitioning, so called *refinements* can be applied. Usually, these are *Kernighan-Lin* based, but the alter-

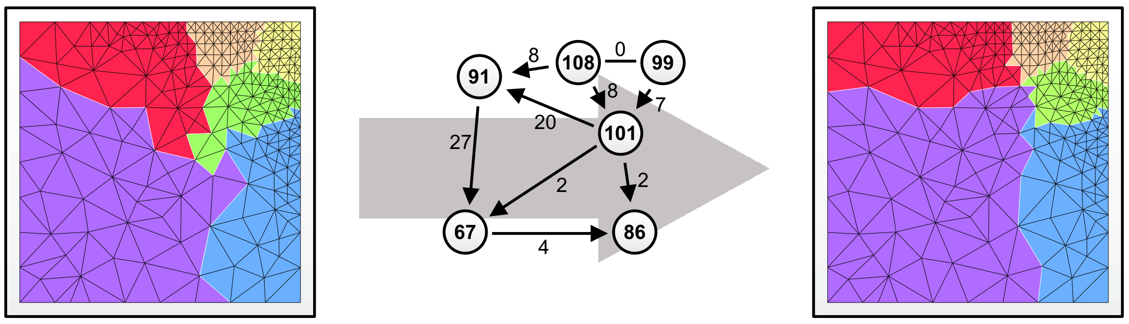


Figure 1.1: After refining the mesh, the computational load is unbalanced (left). A balancing flow is computed (middle) and elements are migrated accordingly (right).

native *Helpful-Set* bisection refinement heuristic often finds solutions with less cut edges. Applying such a refinement inside an hierarchical approach is currently most successful graph partitioning strategy, known as the *Multilevel Paradigm*. The graph partitioning library *Party* proceeds this way and creates its hierarchy according to 2-approximations of a maximum matchings. In chapter 3 we describe our improved implementation of the Helpful-Sets refinement heuristic and compare the obtained results with other state-of-the-art libraries.

Many simulations generate work load dynamically what degrades an existing distribution. Hence, the load must be reassigned during the run-time of the application. The redistribution should keep the costs induced by the task migration as low as possible while still assuring a balanced work load and little communication costs.

An application showing the importance of an efficient load balancing scheme is the *parallel adaptive finite element (FE) simulation*. The involved meshes consist out of several million elements that discretize the geometric space and that are distributed evenly onto the processing nodes. Each processor starts computing independently on its part until the next global communication step is required. Depending on the application, the mesh is refined and coarsened in some areas during the computation which causes an imbalance between the processor loads. For example, exact simulations of turbulences in fluid dynamics depend on such refinements. In these situations, the computational load must be rebalanced as sketched in figure 1.1.

Rebalancing load involves two subproblems: Computing a balancing flow, i. e. determining how much work load needs to be migrated over the communication links, and the choice of the tasks to be placed on or migrated between the processing nodes.

Efficient methods to answer the first question have been developed and are described in chapter 4. In the according model, all calculations can be performed independently from each other, meaning that either no data dependencies between the tasks exist or that their communication costs are negligible. When rearranging the work load, the focus lies on minimizing the number of migrating tasks, or, from the network's point of view, in stressing the communication links as little as possible. An usual assumption is that tasks are splittable arbitrarily. This problem is referred to as *dynamic load balancing*. A flow that is minimal in the $||\cdot||_2$-norm (Euclidean norm) can be computed by solving a system of linear equations. However, this presumes global knowledge of all load values. In contrast, the same solution can be obtained with an iterative diffusion process only utilizing local information, and we depict the *General Diffusion Scheme* and its generalizations. Optimal parameters can be determined for selected processor topologies and we summarize our investigations on the behavior of the scheme in dynamically changing networks.

Once a balancing flow has been computed, most existing repartitioning libraries restrict the vertex movements in the refinement process accordingly. Thus, the refinement procedure that is designed to reduce the number of cut edges locally now becomes also responsible for global decisions like the movement and placement of whole partitions. However, our observations show that though this method, in combination with a graph hierarchy, works well in some cases, it often results in bad partition arrangements. With an increasing number of graph changes and consequent repartitionings, the computed domains tend to become long and thin and eventually fall apart. This behavior also indicates that the commonly used edge-cut metric does not always measures the quality of a partitioning satisfactorily. Several other aspects like domain connectivity or small domain boundaries, which are often desired by the application developers, are not reflected at all. Hence, heuristics that also consider these additional metrics are desirable.

In this thesis we propose a new general graph repartitioning method, but which is designed to balance parallel adaptive numerical simulations. In contrast to existing heuristics, the presented approach does not focus on few cut edges, but essentially computes well shaped domains that closely conform to the informal specifications stated by the developers of numerical simulations.

The new heuristic runs a number of diffusion processes in a learning framework. In each learning iteration, the vertices are assigned to the partitions according to the load distribution of the diffusion. These distributions reflect important information about the graph structure like the connectivity and distances between the vertices and the domains. Hence, the accumulated global knowledge enables the learning process to determine good partition arrangements. To enhance the reliability of the procedure, we modify the General Diffusion Scheme to better meet our special requirements, and analyse the resulting process in chapter 5.

The parallel nature of the diffusion and its integration into the learning framework allow a direct and efficient parallelization. We implement two different parallelization strategies in our repartitioning library *Flux*, which contains additional improvements that further reduce the computation-time. The included experiments show that our new heuristic often finds better solutions than state-of-the-art partitioning libraries, not only concerning the domain shape but also with regard to the number of boundary vertices and the classical edge-cut metric. A new set of benchmarks allows us to observe the behavior of the partitioning libraries over many graph adaptations. Additionally, we have integrated our library into a parallel adaptive simulation environment to study and evaluate the new method. All results confirm the above statement for every single time step. Hence, our new graph partitioning and repartitioning heuristic computes solutions that meet the requirements of parallel adaptive numerical simulations very well.

Publications

Parts of this thesis have been reviewed for and published in the proceedings of the International Conference on Computational Science and Its Applications, 2003, the International Conference on Parallel and Distributed Processing Techniques and Applications, 2003, the 7th International Conference on Parallel Computing Technologies, 2003, the 18th International Parallel and Distributed Processing Symposium, 2004, the 7th International Parallel and Distributed Processing Symposium, 2004, the 17th International Conference on Parallel and Distributed Computing Systems, 2004, the 16th International Symposium on Computer Architecture and High Performance, 2004, the Parallel and Distributed Computing and Networks Conference, 2005, the International Conference on Parallel and Distributed Processing Techniques and Applications, 2005, the 8th International Conference on Parallel Computing Technologies, 2005, European Conference on Parallel Computing, 2005, the 20th International Parallel and Distributed Processing Symposium, 2006, and European Conference on Parallel Computing, 2006.

2 Graph Partitioning

This chapter begins with a definition of the graph partitioning problem, and we refer to a number of theoretical bounds on it. We continue to present relevant heuristic approaches and discuss their properties. The *Spectral Bisection* employs attributes of the algebraic representation of a graph and is a popular method in the process of designing digital circuit layouts. *Linear orderings* enumerate the graph's vertices and assign vertex intervals of appropriate size to the processors. While common orderings are related to coordinates, we introduce an alternative algorithm that better reflects the graph structure. Next, we describe the two local refinement strategies *Kernighan-Lin* and *Helpful-Sets*, which can be applied to improve a given partitioning. Refinement is one of the key components in the *multilevel framework*, together with a matching strategy that creates the involved graph hierarchy. We conclude this chapter with a reference to state-of-the-art graph partitioning and repartitioning libraries that base on this multilevel strategy.

2.1 Problem Definition

Formally the graph partitioning problem can be defined as follows. Given an undirected graph $G = (V, E)$ with vertices V and edges E, a partitioning π of G into P domains is defined as a function

$$\pi : V \to \{1, \ldots, P\}.$$

Hence, the vertices V of the graph are assigned to the P pairwise disjoint parts V_i:

$$V = V_1 \cup \ldots \cup V_P \text{ with } V_i \cap V_j = \emptyset \ \ \forall\, i \neq j.$$

In case of $P = 2$, π is also called *bisection*. The balance (or imbalance) of a partitioning $\mathrm{bal}(\pi)$ reflects the largest size of a partition in relation to the average:

$$\mathrm{bal}(\pi) = \frac{\max_i\{|V_i|\}}{|V|/P}.$$

π is totally balanced, if $\text{bal}(\pi) = 1$, which is only possible, if P divides $|V|$. The edge-cut (or cut) is defined as the number of edges incident to two vertices from different parts

$$\text{cut}(\pi) = |\{\{v, u\} \in E : \pi(v) \neq \pi(u)\}|\,.$$

Now, the graph partitioning problem consists in finding a totally balanced partitioning with the smallest possible edge-cut. It is known to be NP-complete [GJ79], hence optimal solutions for large input instances cannot be determined in a reasonable amount of time.

In case of a bisection, the minimal possible cut of a balanced partitioning is called *bisection width*

$$\text{bw}(G) = \min_{\pi}\{\text{cut}(\pi) : \pi \text{ is a totally balanced bisection of } G\}.$$

Already for regular graphs, the graph bisection problem as the simplest form of the graph partitioning problem is NP-complete in [BCLS87].

2.2 Analytical Bounds

Several analytical bounds on the graph bisection width are known. There exists an algorithm which calculates a cut-size that differs from the bisection width by not more than a factor of $\mathcal{O}(\sqrt{|V|} \cdot \log(|V|))$ [FKN00]. This first sub-linear approximation factor has been improved to $\mathcal{O}(\log^2(|V|))$ [FK02].

Analytical results on regular graphs show that almost every large d-regular graph $G = (V, E)$ has a bisection width of at least $c_d \cdot |V|$ where $c_d \to \frac{d}{4}$ as $d \to \infty$ [CE88, Bol88]. These bounds can be improved for small values of d. Almost every large 3-regular graph has a bisection width of at least $\frac{1}{9.9}|V| \approx 0.101|V|$ [KM92, KM93]. On the other hand, all sufficiently large 3-regular graphs possess a bisection width of at most $\frac{1}{6}|V|$ [MP01]. Almost all large 4-regular graphs have a bisection width of at least $\frac{11}{50}|V| = 0.22|V|$ [Bol88], while it is at most $\frac{2}{5}|V|$ in case of sufficiently large 4-regular graphs.

Some approaches calculate lower bounds on the graph bisection width. These bounds can be used to evaluate the quality of the existing upper bounds as well as to speed up Branch & Bound strategies determining the bisection width of moderately-sized graphs. One lower bound of the bisection width based on a routing scheme for all pairs of vertices [Lei92]. A small congestion of the routing scheme leads to a large lower bound. Lower bounds on the bisection width can also be derived from algebraic graph theory by relating the bisection problem to an eigenvalue problem. It is well known that the bisection width of a graph $G = (V, E)$ is at least $\lambda_2|V|/4$ with λ_2 being the second smallest eigenvalue of the Laplacian matrix of G. This spectral bound is tight for some graphs [BEM+04].

Furthermore, the structure of an optimal bisection can be used to derive improved spectral lower bounds on certain graph classes [BEM+04]. For some classes of d-regular graphs one can prove an improved lower bound on the bisection width of roughly $(d/(d-2)) \cdot (\lambda_2|V|/4)$. Furthermore, one can prove a lower bound of $(10 + \lambda_2^2 - 7\lambda_2)/(8 + 3\lambda_2^3 - 17\lambda_2^2 + 10\lambda_2) \cdot (\lambda_2|V|/2)$ for the bisection width of all sufficiently large 3-regular graphs and a lower bound of $(5 - \lambda_2)/(7 - (\lambda_2 - 1)^2) \cdot (\lambda_2|V|/2)$ for the bisection width of all sufficiently large 4-regular graphs. These lower bounds are better than the classical bound of $\lambda_2|V|/4$ for sufficiently large graphs and are applicable to Ramanujan graphs [Chi92, Mor94]. Any sufficiently large 3-regular Ramanujan graph has a bisection width of at least $0.082|V|$ while sufficiently large 4-regular Ramanujan graphs have a bisection width of at least $0.176|V|$. These values are the best lower bounds for explicitly constructible 3- and 4-regular graphs [MP01].

Though these bounds are of high theoretical interest, they are far from acceptable for real applications. Furthermore, the algorithms behind them are often very complicated and are not suitable to design fast and efficient general graph partitioning algorithms. Due to the problem's practical importance, a number of heuristics have been developed that usually provide 'good enough' solutions quickly.

2.3 Global Methods

Graph partitioning heuristics that operate directly on the input-graph are referred to as *global methods*. Due to the large problem sizes of common applications, it is difficult for a global method to be fast and to deliver good solutions. Apart from simple greedy algorithms, a few applicable approaches have been developed.

2.3.1 Spectral Bisectioning

A popular partitioning method is the Spectral Bisection which works on the Laplacian matrix $\mathbf{L}$, an algebraic representation of the graph. $\mathbf{L}$ contains the vertex degrees on the main diagonal, the off-diagonal entries $\mathbf{L}_{(ij)}$ are set to -1, if the edge (i, j) exists and 0 otherwise.

The bisection is based on the eigenvector e corresponding to the the second-smallest eigenvalue λ_2 of $\mathbf{L}$. e is called *Fiedler Vector* [Fie73, Fie75] and expresses the graph's algebraic connectivity. The median m of all e's components is determined and the vertices v of the graph are split into $V_1 = \{v \in V : e_v < m\}$ and $V_2 = \{v \in V : e_v \geq m\}$ [PSL90]. According to Fiedler's results, at least one of the subgraphs is connected if G is connected. The results of the Spectral Bisection are usually fairly good considering that it is a global approach, but local improvements are still possible. Furthermore, the computation of λ_2 is quite costly for large graph instances. An extension of this approach utilizes more eigenvectors and allows to divide the graph into more than 2 domains [HL93].

2.3.2 Linear Orderings

Some applications request to partition structurally identical graphs. This is necessary if computational load changes on the processing nodes occur rapidly and unpredictably or if the number of available processors changes frequently. In these cases the load has to be rebalanced very quickly to avoid the overall runtime being dominated by the load-balancing algorithm.

An extremely fast approach to balance structurally identical graphs consisting of vertices with varying vertex weights is given by *implicit partitioning*: Initially, the graphs vertices are rearranged according to a certain scheme, such that the following partitioning problems are reduced to simply splitting the node list into intervals of appropriate size. To keep the edge-cut reasonable low and the partitions compact, the node ordering has to preserve locality.

Finding a linear ordering of a graph is a combinatorial problem and has been studied extensively with respect to various objective functions. There exist a number of theoretical bounds, approximation algorithms, and heuristics [DPS02]. Most of the considered metrics are not suited to determine good orderings for implicit graph partitioning. Even the wire-length metrics (also denoted as edge sum, minimum 1-sum, or bandwidth sum) that does reflect some kind of locality, usually does not lead to a small global edge-cut of the implicit partitionings. For example, optimizing this metric always results in placements of the first and last vertex of the curve far away from each other, what is neither required nor helpful for an implicit partitioning.

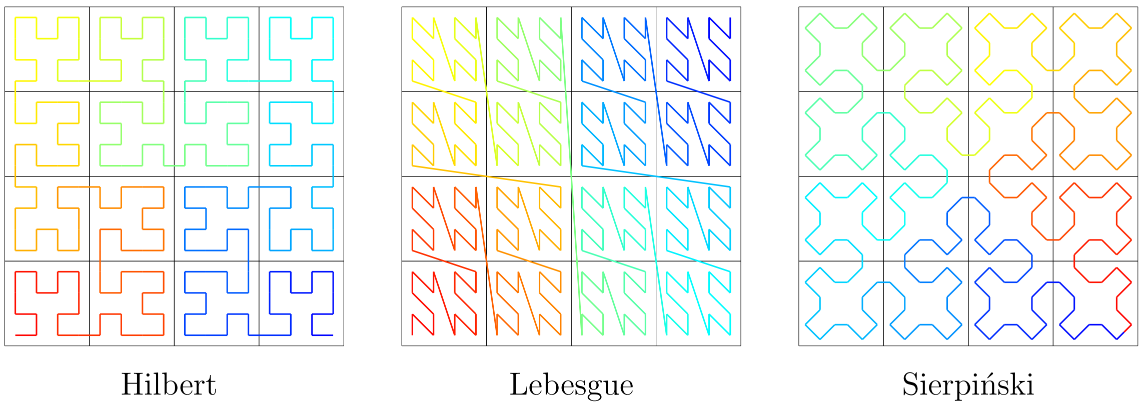


Figure 2.1: Structure of selected space-filling curves in the 16×16 grid.

Space-Filling Curves

Space-filling Curves (SFC) are geometric representations of bijective mappings $M : \{1, \ldots, N^m\} \rightarrow \{1, \ldots, N\}^m$. The curve M traverses all N^m cells in the m-dimensional grid of size N. They have been introduced by Peano and Hilbert in the late 19th century [Hil91, Sag94]. Three well known types are sketched in figure 2.1.

Although space-filling curves were initially defined on grids, they can also be used to determine node orderings in unstructured meshes. The graphs' vertices are separated recursively and aligned according to the underlying curve structure. The recursion is repeated until each subpart contains a single vertex. Note that this computation requires vertex coordinates. From the definition of the curves follows that the corresponding separators are $n-1$-dimensional hyper-planes in the n-dimensional space. The locality preserving properties ensure that each interval of the curve forms a partition with a to some extend limited number of edges to the remainder of the graph.

However, this only holds as long as the given mesh structure allows a suitable embedding of the curve, like it is the case for regular grids. If this is not possible, the purely geometric separation destroys the locality properties of the ordering which are necessary to find good graph partitioning solutions. Comparisons between advanced graph partitioning heuristics and space-filling curves [HD00, SKK02, SW03] reveal that the advanced partitioning tools determine solutions with a lower edge-cut. Especially in some configurations that often occur in FE mesh partitioning, space-filling curves tend to produce partitionings of bad quality. The most important cases are: The number of requested partitions is small, the mesh contains holes (un-discretized

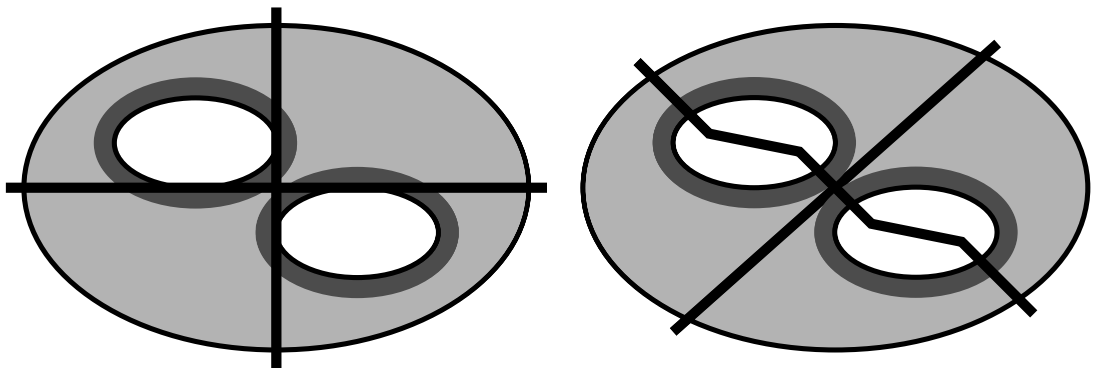


Figure 2.2: Defining a locality preserving curve is often difficult if the graph separation is based on geometric information (left). A separation based on graph partitioning can overcome this problem (right).

regions embedded in the mesh), or unevenly refined regions exist whose expansion is of lower dimension than the overall problem (e.g. a 2-dimensional area within a 3-dimensional FE mesh). This is a consequence of ignoring the provided connectivity information but only relying on coordinates.

The example given in figure 2.2 (left) sketches a graph with two holes in a refined area. The graph is separated according to a SFC based ordering. Here, large parts of the cuts traverse deeply refined areas. Since boundaries of partitions based on orderings often coincide with the separators of high levels, the resulting implicit partitioning quality can be expected to be quite poor. Furthermore, a curve has to cover every vertex in a region before entering the next one. Hence, in the shown case, the curve loses much of its locality.

Graph-Filling Curves

In the following, we present a more suitable method to subdivide the vertices and arrange them in a locality preserving way than it is possible with SFCs. The main idea is to construct a curve which does not rely on geometry based separators and static ordering rules which can hardly be embedded into unstructured meshes.

In figure 2.2 (right) the sketched separators are based on a high quality 4-partitioning. Assuming that they are part of the final solution, it becomes clear that now a partitioning with a much lower edge-cut can be found than in case of the coordinate based approach.

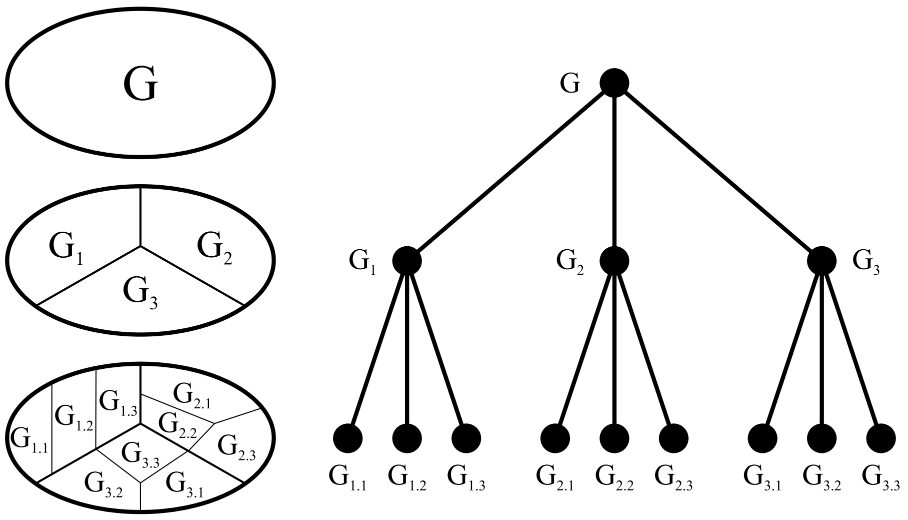


Figure 2.3: Phase 1: The construction of a hierarchy based on a partitioning algorithm. In every level, the (sub-)graph is divided into 3 parts (left). The resulting tree hierarchy is shown (right).

To find a suitable linear locality preserving ordering, we propose the following two-phase algorithm. In the first phase, the graph is recursively partitioned and the corresponding hierarchy tree is constructed. In the second phase, an appropriate rearrangement within the tree is performed, which is supposed to improve the locality of the ordering.

In the partitioning phase, each recursion divides the current subgraph into k parts, where k is a small constant. This phase continues until only one vertex in a subgraph remains. The first two levels of this process are illustrated in figure 2.3 for $k = 3$.

Traversing the constructed hierarchy defines the order of its leaf nodes representing the graph's vertices. The second phase of the algorithm tries to optimize this ordering by suitably rearranging the successors of each node in the tree. This means that we allow a locally restricted reordering of nodes on the same level. In the current implementation, the vertices are ordered such that the number of edges between consequent subgraphs is maximized.

Since our aim is to determine orderings that induce a small global edge-cut, we propose to link partitions with a large common boundary. This increases the probability that the curve will still be connected in the lower levels of the tree's hierarchy. A possible rearrangement in the hierarchy tree is sketched in figure 2.4. Starting with the solution given in figure 2.3, the nodes representing subgraphs G_2 and G_3

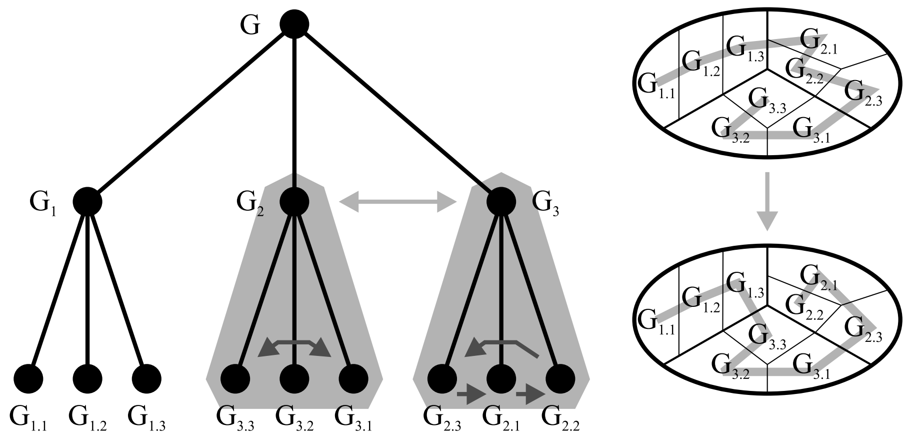


Figure 2.4: Phase 2: Traversing the hierarchy defines a curve. Its locality might be improved by exchanging subgraphs (and their successors) of the same levels.

are exchanged, as well as the nodes $G_{3.1}$ and $G_{3.3}$. The nodes $G_{2.i}$ are shifted to the right, respectively. Note, that all rearrangement operations involve nodes with the same successor only. Hence, the previously defined orderings in higher levels remain unchanged. On the right hand side of the figure, the old (top) and the new (bottom) curve is plotted in the sketched graph.

From the description of the algorithm follows that there is an obvious dependency between the choice of the parameter k and the number of possible rearrangements in the second phase. A small k leads to a high grade of convolution while a large k increases the flexibility in the reordering of the hierarchy. In the case of bisections, there are obviously only two possibilities to traverse the successors of each node.

Due to the relationship to the graph's structure rather than to its geometric shape we call the generated traversals *graph-filling curves* (GFC).

To provide a detailed example, we analyze the space-filling and graph-filling curves on the relatively small 'airfoil.1' mesh. This unstructured mesh consists of 4253 vertices and 12289 edges and contains holes with a very fine discretized region around them. Figure 2.5 shows the computed curves and the partitioned graph. The top row displays the obtained SFC and GFC orderings while the bottom illustrations present the resulting implicit 5-partitionings. The orderings are indicated by a curve ranging from red to blue. For the SFC approach we applied the Hilbert curve while the GFC is based on recursive bipartitioning combined with the proposed ordering algorithm.

The SFC ordering starts in the lower left corner and follows the Hilbert curve to the lower right corner. The course of the separators of the higher levels is well visible, especially the first vertical one which results in only one connection between the left and the right part of the curve. As one can see, there are a lot of curve segments crossing the holes in the graph. Partitions based on such an ordering will contain vertices from both sides of the hole as demonstrated below for the implicit 5-partitioning. As expected, the upper left partition contains vertices from above as well as from below the large hole. Furthermore, the course of the horizontal separator of the highest level leads to a very large cut between both partitions on the left hand side. A closer view reveals similar cuts between other partition pairs.

The GFC ordering starts close to the right end of the large hole in the area of fine discretization. The curve continues below the smallest and the left hole. It surrounds the group of holes in counter clockwise order and ends close to its starting point. On a first view, the curve seems quite convoluted. This effect emerges from a missing local optimization phase.

A closer look reveals that these obvious misalignments of the curve almost always affect only vertices close together. Hence, it only influences the quality of an induced partition at its entrance and exit to the partition. Furthermore, one can see that the curve crosses the holes only once. The resulting 5-partitioning is shown in the lower right illustration. Apart from some single vertices, only one partition is disconnected and split into two parts. The partition in the lower left corner has additional vertices in the region between the two smaller holes.

With a global edge-cut of 318, the implicit 5-partitioning based on the GFC approach is much better than the one obtained according to the Hilbert curve, which has an edge-cut of 624. As a comparison, elaborated graph partitioning heuristics like Party and Metis are able to find separators with an edge-cut of 202 and 231, respectively. Although the latter values are smaller, the numbers give an indication of the improvement due to the new linear ordering approach.

The experiments presented in the following are based on a set of well known 2- and 3-dimensional FEM graphs. The 2-dimensional graphs ’biplane.9’ and ’shock.9’ have a grid-like structure in which all edges are axis aligned. ’crack’ and ’big’ are much more unstructured with arbitrarily oriented edges at triangular elements. All of the 3-dimensional benchmark graphs ’rotor’, ’wave’ and ’hermes’ base on a mesh discretization into tetrahedrons. Especially ’rotor’ has a widespread zone consisting of very fine elements.

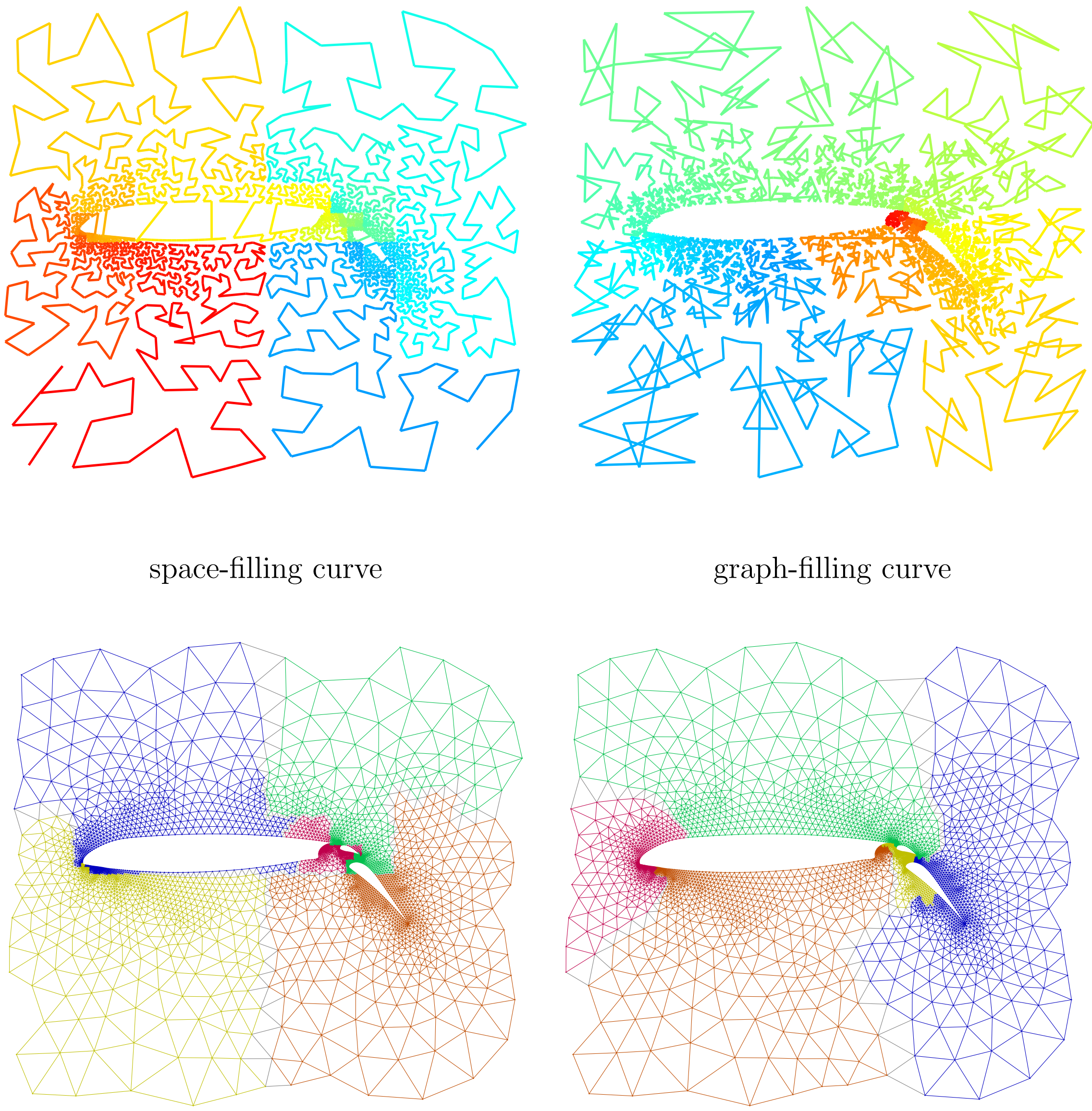
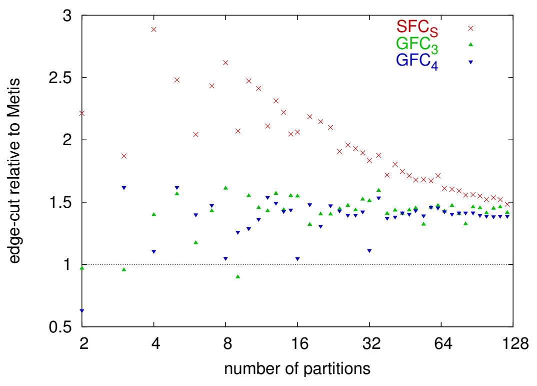


Figure 2.5: SFC and a GFC based implicit 5-partitionings in comparison.



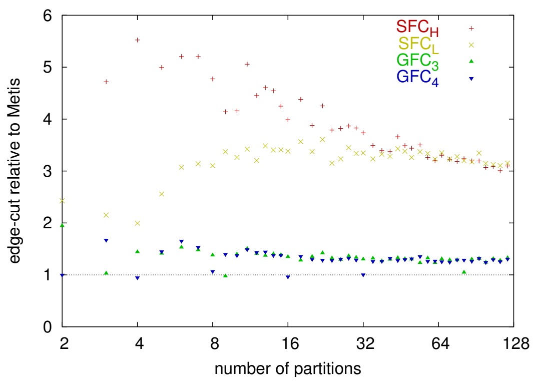


Figure 2.6: Comparing SFC vs. GFC orderings according to the edge-cut for the graphs 'big' (left) and 'rotor' (right).

In figure 2.6, the edge-cut of the implicitly generated partitionings based on SFC and GFC orderings in the meshes 'big' and 'rotor' are displayed. As in all following experiments, the number of created partitions ranges from 2 to 128. To be able to estimate the benefit of GFC as well as the overall quality, the obtained edge-cuts are normalized to those calculated by Metis. Note, that in case of 'big' the SFC ordering is based on the Sierpiński curve, since further examinations have revealed that this ordering leads to the best solutions in unstructured meshes. The GFC ordering depends on the parameter k for the recursive partitioning in phase 1. The results for $k = 3$ and $k = 4$ are labeled as GFC_3 and GFC_4, respectively. For the 'rotor' mesh, the SFC ordering is based on the 3-dimensional variants of the Hilbert and the Lebesgue curve, labeled by SFC_H and SFC_L, respectively.

It becomes obvious that the GFC approach is superior to SFC over the whole range of examined partition numbers. Remarkable is the fact that in contrast to SFC, the partition quality induced by the GFC ordering is less influenced by the number of desired partitions. Most of the solutions have a quality better than factor 1.5 in relation to Metis. That means that the obtained implicit partitioning has an increase in edge cut of less than 50% compared to the elaborated graph partitioning library. Values lower than 1 arise from the good solutions delivered by the Party library, which we apply as the underlying graph partitioner. The GFC profits directly from these separators which are better than those of Metis.

As mentioned above, the quality of resulting partitionings depends on the right choice of k. Especially for a small number of partitions, the edge-cut of the obtained results befalls a high variation. For example, the GFC_4 ordering leads to good results in case of 2- and 4-partitionings but is bad in case of the 3-partitioning. In this case GFC_3 performs much better. Although this behavior has been expected, it is interesting to see that the difference almost disappears for larger number of partitions. Further experiments with other values of k have confirmed this observation.

The quality of partitions induced by GFC with parameters $k = 3$ and $k = 4$ are much closer to the results of Metis. 79% ($k = 3$) and 84% ($k = 4$) of the achieved edge-cuts are below the factor 1.4. The averages in normalized quality are 1.33 ($k = 3$) and 1.30 ($k = 4$). This means that we can expect an increase in edge-cut of just one third compared to Metis, instead of a factor of 3 to 4. The variation of the values is much smaller compared to SFC orderings and again less dependent on the number of partitions. Furthermore, different values for the parameter k have less influence while the choice of the space-filling curve (Lebesgue vs. Hilbert) has dramatic consequences on the partition quality, especially for small partition numbers. Most of the values obtained for both SFC variants lie in the range of 3 to 4. The Hilbert curve induces even worse results in case of small partition numbers with a peak (8.7) in case of the bisection. Note that this value is omitted to allow a more concise scaling. For partitionings into 5 or less parts the Lebesgue curve reaches better qualities, but the overall behavior of both geometric schemes is not acceptable for most applications.

The experimental results for all graphs of our test-set are summarized in table 2.1. The edge-cuts achieved by Metis, SFC, and GFC are given for the 4-, 16-, and 64-partitioning. For the unstructured meshes 'crack' and 'big', the ordering was generated according to the structure of the Sierpiński curve, while all others follow the Lebesgue curve or its 3-dimensional extension. Among the traditional SFC orderings, these settings achieve the best or close to the best results. The GFC approach is based on the parameter $k = 3$ for the recursive partitioning because $k = 2$ or $k = 4$ will automatically induce high quality P-partitionings if P is a power of 4.

The table shows that in almost all cases GFC outperforms SFC, in many cases even dramatically. Only in two settings (64-partitioning of 'crack' and 16-partitioning of 'shock.9') the edge-cut of the partitioning induced by a SFC ordering is slightly better than the one of the solution based on GFC. Compared to Metis, the edge-cut of solutions computed with SFC is in average about 2.1 times as high ($[1.35, \ldots, 3.82]$). Applying GFC instead, this factor can be decreased down to a value of 1.4 ($[1.26, \ldots, 1.86]$).

Table 2.1: Comparison of edge cuts: Metis, SFC (Lebesgue), and GFC.

	4-partitioning			16-partitioning			64-partitioning		
Graph	Metis	SFC	GFC	Metis	SFC	GFC	Metis	SFC	GFC
crack	408	1066	648	1218	2013	1831	2781	3927	3944
big	344	993	481	1099	2267	1701	2843	4854	4166
biplane.9	193	468	256	800	1235	1086	1906	2888	2653
shock.9	449	637	632	1208	1675	1700	2902	3915	3832
brack	3493	7696	6508	13225	28086	18696	29432	61054	39755
rotor	8829	27976	12726	24477	93526	32996	52190	182657	67499
wave	21682	32101	27349	48183	78592	65050	94342	138787	120314
hermes	46258	121152	65364	119219	266479	154419	241771	473081	322317

Note, that the computation time of the GFC algorithm is by far longest. This is due to the multiple invocation of the graph partitioner as well as the additional rearrangement phase. However, the ordering for a graph has to be determined only once and all implicit partitionings can thereafter be constructed in almost no time.

Our experimental results show that in general the new ordering approach produces partitionings with a quality much closer to the best known solutions than its geometric counterparts, indicating that the GFC approach is superior to SFC over the whole range of examined partition numbers. Remarkable is the fact that in contrast to SFC, the partition quality induced by the GFC ordering is less influenced by the number of desired partitions. Compared to the graph partitioning library Metis, the number of cut edges obtained by the implicit partitioning is now usually less than 50% higher.

2.4 Local Heuristics

While global heuristics compute a partitioning from scratch, local heuristics improve a given partitioning. This is achieved by local rearrangements that exchange vertices or sets of vertices between the partitions, also called refinements.

2.4.1 Kernighan-Lin

The Kernighan-Lin (KL) heuristic [KL70] is one of the earliest graph bisectioning heuristics and has been developed to optimize placements of electronic circuits. The original algorithm by Kernighan and Lin is based on the exchange of vertex pairs.

The decision of which vertices to be exchanged is based on their gain. Denoting the edge-weight of an edge (u, v) as $w(u, v)$, for the current bisection π

$$\mathrm{ex}(v) = \sum_{(v,u)\in E, \pi(v)\neq\pi(u)} w(v, u) \ , \qquad \mathrm{in}(v) = \sum_{(v,u)\in E, \pi(v)=\pi(u)} w(v, u)$$

state the external and internal degree of a vertex v. The gain is then defined as the edge-cut reduction that would occur if v was moved to the other partition:

$$\mathrm{gain}(v) = \mathrm{ex}(v) - \mathrm{in}(v).$$

Thus, if v and u are two vertices, one from each partition, then swapping them reduces the edge cut by $\mathrm{gain}(u) + \mathrm{gain}(v) - 2w(v, u)$, where $w(v, u)$ is the edge-weight of (v, u), or 0 if this edge does not exist.

To escape local minima, the KL algorithm performs several passes. In each pass, every vertex is allowed to change partitions once. The vertex pairs are sorted according to their gain and the ones with the highest gain are migrated greedily. Since many vertex pairs with the same gain exists, several scanning strategies have been investigated [Dut93]. The edge-cut reduction is recorded, and finally migrations are unrolled to the best recorded solution that occurred during a pass. To speed up the process, a pass can be aborted if the edge-cut reduction falls below a certain value. This value has to be chosen with care, since if the passes are exited to early the algorithm will be stuck in local minima.

The KL algorithm has been modified [FM82, Kri84] such that a single vertex is moved followed by a move of another vertex in the reverse direction. Doing so, the time complexity of a pass has been reduced from $\Theta(|V|^2 \log |V|)$ to $\Theta(|E|)$. However, it should be noted that swapping vertex pairs gives a better edge-cut improvement. Furthermore, the stated run-times only hold if the edges are either unweighted or limited to some discrete values such that a bucket sort mechanism can be applied, otherwise it will rise to $\Theta(|V| \log |V| + |E|)$ [Dut93].

Further extensions of the KL algorithm allow to refine the edge-cut in case of more than 2 partitions [HL95]. Note, that a k-way refinement is significantly more complicated than a 2-way refinement because vertices can be moved to a number of partitions which enlarges the optimization space combinatorially. Hence, some simplifications have been invented that still result in a powerful heuristic [KK97].

2.4.2 Helpful-Sets

Just as KL, the Helpful-Set (HS) refinement heuristic is based on local search. Beginning with a given initial bisection π, it tries to reduce the edge cut with the help of local rearrangements. However, their choice is the main difference to KL since the heuristic does not only migrate single vertices but vertex sets. It starts to search for l-helpful sets, that is a subset of nodes from either partition V_1 or V_2 that decreases the edge cut by l if moved to the other partition. If such a set is found, it is migrated and the algorithm then tries to find a balancing set in the opposite partition that eliminates the caused imbalance and does not increase the edge cut by more than $l-1$. Moving this set results in a total edge cut reduction of at least 1. The whole process is repeated until no more improvements can be made.

The heuristic is derived from theoretical analysis on regular graphs. As long as the edge-cut is above a certain value, one can show that a helpful set and a balancing set do exist. Utilizing this knowledge provides constructive bounds for the bisection width [HM91, MD97]. If $G = (V, E)$ is a d-regular graph with even d, $d \geq 4$ and the number of vertices in the graph is large enough, the bisection width $\mathrm{bw}(G)$ of G is bounded by

$$\mathrm{bw}(G) \leq \frac{d-2}{4}|V| + 1.$$

The Helpful-Set heuristic [DMP95, MPD00, MS04] is the algorithmic result of the above theorem and includes several generalizations like the handling of graphs with arbitrary vertex degrees and graphs with vertex and edge weights. It has been successfully implemented in the graph partitioning library Party [Pre00]. Our new enhanced version is presented in chapter 3.

2.5 The Multilevel Scheme

A breakthrough in the field of graph partitioning is the introduction of the multilevel scheme [HL95, SKK97]. Instead of immediately computing a partitioning for the large input graph, vertices are contracted and a smaller instance with a similar structure is generated. On this graph, the partitioning problem is solved applying a global heuristic. Due to the reduced size it is easier to find sufficiently good solutions. Afterwards, vertices of the original graph are assigned to partitions according to their representatives in the smaller instance. On every level, the obtained solution is further enhanced by a local refinement heuristic.

To obtain a global solution on the smaller instance, the described process can be repeated recursively, until in the lowest level only a very small graph remains. Hence, a very basic global heuristic can be applied and it can even be omitted if the number of remaining vertices equals the requested number of partitions. Using a level hierarchy allows to determine good solutions quickly, even for large instances, since the refinement process always starts with a good and quite balanced solution from the next lower level. Hence, no major partition rearrangements are necessary.

Summarized, a multilevel algorithm consists of two or three important tasks: An algorithm deciding which vertices are combined in the next level, a global partitioning algorithm applied in the lowest level if necessary, and a local refinement algorithm that improves the quality of a given partitioning.

2.5.1 Matching Algorithms

To create a smaller, similar graph for the next level of the multi-level scheme, usually a matching algorithm is applied. The matched vertices are joined and the edges combined accordingly. A matching algorithm for multilevel partitioning is supposed to be fast and to have a high matching cardinality and matching weight. Because of the time constraints, the calculation of a maximum cardinality matching or even a maximum weighted matching would be too time consuming. Therefore, fast algorithms calculating maximal matchings are applied.

A very simple algorithm is based on a greedy strategy always adding the next heavier free edge to the matching [Avi83]. A number of alternative approaches have been proposed [HL95, KK98b, KK99], but do not provide any quality guarantee. In contrast, the LAM algorithm [Pre99, MPD00] guarantees to find a maximum weighted matching with at least half the weight of the optimal solution. It has been the first linear time approximation algorithm for maximum weighted matchings. The discovery of this algorithm initiated a number of improvements [DH03b, DH03a] which simplify the algorithm and improve the approximation factor. All of them follow the same strategy: Starting with an initial empty matching, the vertices of the graph are visited in a specific order. For each visited vertex v, it is checked if v is free, i. e. can still be matched, and if v is adjacent to at least one unmatched vertex. If v is free and all neighbors are already matched, v remains free. Otherwise, if v is unmatched and at least one free neighbor exists, the edges to free neighbors are rated and an edge with highest rating is added to the matching.

2.5.2 Partitioning Libraries

The efficiency of a graph partitioning heuristic in terms of run-time and solution quality strongly depends on its implementation. Besides specialised realizations inside some applications, there exist several general purpose libraries. They all base in the multilevel scheme, but vary in their implementation details and parameter settings. The most prominent examples are:

Chaco [HL94] is one of the first freely available library. It includes several variations of the Spectral Bisection approach as well as a multi-dimension spectral partitioner. Both can be applied within the multilevel approach. Additionally, the KL refinement heuristic is implemented.

Jostle [Wal02] applies the KL heuristic within the multilevel scheme, and additionally includes a simple technique that addresses the domain shape. Additionally, it contains a balancing strategy that can handle multiple vertex weights.

Metis [KK98a] is a family of programs for partitioning unstructured graphs and hypergraphs and computing fill-reducing orderings of sparse matrices. It consists of different frontends: pMetis and kMetis are recursive and direct graph partitioners. hMetis is a program for partitioning hypergraphs such as those corresponding to VLSI circuits. Furthermore, methods for multi-constraint and multi-objective partitioning are implemented in some of the packages.

Party [Pre98] applies the Helpful-Set heuristic to find local improvements. Furthermore, it is the first library that employs an approximation scheme of a maximum weighted matching to create the graph hierarchy. Since the publication of version 2.001 [Pre00], we have implemented several enhancements addressing the solution quality and the library's reliability as described in chapter 3.

WGPP [Gup96] is a package for partitioning graphs and for computing fill-reduced orderings of sparse matrices for their direct factorization. It generates multiple coarsenings of the same graph and selects the best one when the refined graph becomes large enough and applies some additional preprocessing.

2.6 Graph Repartitioning

Each time the computational mesh in a dynamic application is adapted, the vertices representing the workload have to be redistributed between the processing nodes. Here, besides the balancing and a small edge-cut, the additional objective to keep the number of migrating vertices small has to be considered.

A first idea to address this problem is to compute a partitioning from scratch and then map the old partitions onto the new ones such that the migration is small [SKK97]. However, this still causes many vertices to change, and less migration occurs if the partitioning process takes the current distribution into account.

Better results can be obtained by adopting the multilevel scheme. The global partitioning step on the lowest level is replaced by a vertex assignment that reflects the current vertex distribution of the original graph. To obtain such a unique assignment, the matching can be restricted to join only vertices of the same partition. Depending on how many levels are generated, the assignment is more or less accurate, and one can either target on a small edge-cut or little migration.

Alternatively, it is possible to focus on the overall goal, a short application runtime. Plum [BO98] globally repartitions the computational mesh and determines whether rebalancing the workload would reduce the total execution time. If an improvement in the load balance can be achieved, Plum utilizes one of its several remapping algorithms to minimize the required data movement. The library MinEX [DHB02] optimizes Plum's objective inside a KL refinement. These approaches counter the possibility that perfectly balanced loads with minimal communication can still incur excessive redistribution costs for adaptive applications.

If the graph is already distributed on the processing nodes, it is desirable to apply a distributed repartitioning algorithm. Often, due to memory requirements, it is not even possible to assemble the whole graph on a single node.

However, although there exist a large number of sequential libraries, only a few parallel implementations are available. This is due to the complexity involved in parallel programming. Furthermore, the mostly KL based refinement heuristics are basically of sequential nature, hence modifications are required that introduce new limitations. The most popular distributed libraries are the parallel versions of Metis [KK96, KK98a] and Jostle [WCE97, Wal00, WC00]. These tools basically apply the same techniques as their sequential counterparts, are quite fast and deliver solutions of acceptable quality for most applications.

3 The PARTY Graph Partitioning Library

This chapter describes the graph partitioning library Party. The library is based on the multilevel strategy and applies the LAM matching heuristic as well as the Helpful-Set bisection refinement presented in the last chapter.

First, we introduce a novel evaluation scheme which increases the significance of the information obtained in run-time experiments. Analyzing the gathered data, we have been able to improve the matching strategy and especially the Helpful-Set implementation of which we provide a detailed description.

Using a common set of graph instances, we then compare Party's original [Pre00] and our latest implementation with the state-of-the-art graph partitioning packages Metis and Jostle. In contrast to other evaluations, we also record the quality deviation reflecting the library's reliability. While usually solutions with almost equally sized partitions are requested, we additionally examine the behavior under less restricted balance constraints. Applying bisections recursively, Party is able to partition a graph into more than two parts. We conclude this chapter with the results of extensive tests covering the range of two to 50 partitions. It turns out that Party's new implementation is often able to outperform direct partitioning libraries like Metis and Jostle.

3.1 An Improved Evaluation Scheme

Due to the complexity of the graph partitioning problem, experiments are the usual way to compare and evaluate different heuristics. These tests are performed on an agreed set of representative graph instances [Wal]. However, using a fixed set of graphs to test a partitioning library includes some drawbacks. Even if not intended, it is easy to adopt an algorithm to the test set, meaning that the results become better for the selected graphs but deteriorate for others. During the development

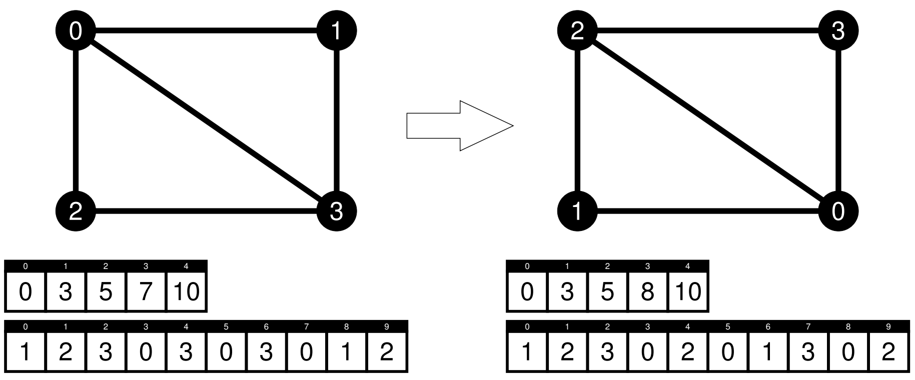


Figure 3.1: The original graph and its data structure (left) is transformed by a vertex permutation $\sigma = (2, 3, 1, 0)$ into the new graph (right).

of the Helpful-Set heuristic, we have experienced this several times. To overcome this problem, we first enlarge the test set. This makes the undesired adoption more unlikely, but still the number of available instances remains quite limited. Another approach we found is based on permutation. It should be mentioned that generating random graphs is no solution to the problem since their structure and their properties are completely different to those of 'real world' problems. However, a graph can be partly randomized maintaining its structure. This is done by permuting its vertices.

Figure 3.1 shows an example of how this construction works. First, a random permutation σ of numbers 0 to $|V| - 1$ is determined. σ is then used to map each vertex v of the original graph G to vertex $\sigma(v)$ in the new graph G'. Next, the newly generated sequence of G's vertex numbers is sorted, changing the vertices' order. After that, the edges are adopted to the new graph by transforming an edge (v_i, v_j) into edge $(v_{\sigma(i)}, v_{\sigma(j)})$. Last, the outgoing edges of each vertex are sorted according to their destination's vertex number. It is obvious that the remaining graph G' has exactly the same structure and properties than the original G. However, in our experiments it is shown that the influence of the permutations on the partitioning results is surprisingly high.

To make use out of this observation, we create a new evaluation scheme, based on the described technique. With each graph in our test set, we performed 100 runs. The first run consists of partitioning the original, unchanged graph, while for all following runs the graphs vertices are permuted. We are aware that this scheme does not solve the evaluation problem completely since an algorithm can still be tailored for special graph types. But at least this becomes more difficult

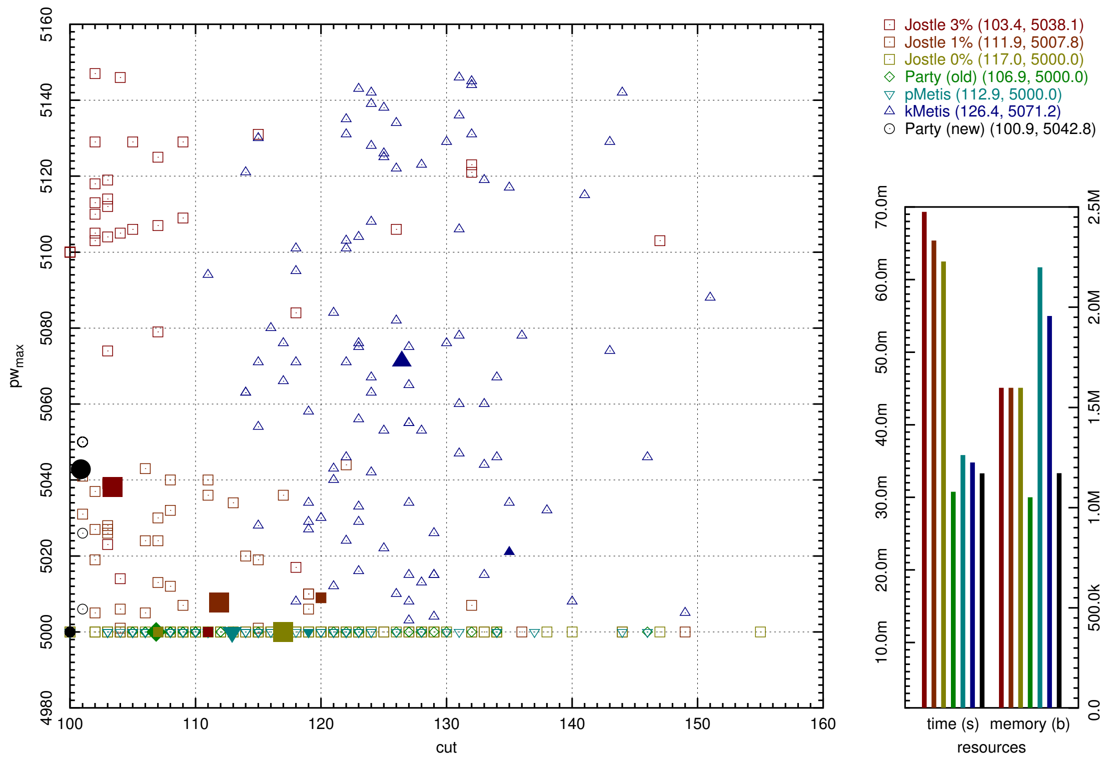


Figure 3.2: Numerical results computing bisections of the 100×100 grid

for single test graphs that now represents a class of graphs. Last but not least, the method delivers some data about the variation of the solution quality. One can now determine if an algorithm does always find solutions of about the same quality or if they highly differ to each other, supplying a measurement for reliability. Thus, we believe that the permutation scheme makes comparisons more meaningful.

The results obtained in the experiments can be displayed in a chart generated by a script. Figure 3.2 gives an example. It shows the results we obtain when bisecting the 100x100 grid, which we choose because optimal solutions (with edge cut 100) are known. The left part shows the bisection quality. The edge cut is displayed on the x-axis while the y-axis displays the weight of the largest partition ($\mathrm{pw}_{\mathrm{max}}$), that is the balance. Every mark of each type represents the result of one of 100 runs for each of the libraries, respectively. Furthermore, the first run with the genuine, unchanged graph is represented by a solid mark, while the average of all 100 runs is displayed by a large solid mark. The right part shows the required resources. Since these do not depend much on the permutation, only the average time and memory consumption from all runs for this graph is displayed.

In figure 3.2, the yellow, orange, and red squares display the results obtained using Jostle with 0%, 1% and 3% allowed imbalance, respectively. One can see that the average edge cut computed gets smaller with a higher imbalance allowance. The average solution for 3% allowed imbalance is of a good quality (103.4), and the figure reveals that there is some variation due to the randomization. This variation is even higher for Metis, whose results are displayed with the upper light (pMetis) and lower dark (kMetis) blue triangles. Especially for kMetis, achieving an average edge cut of 126.4, the variation is very high, both for the edge cut and for the balance. While the results obtained by pMetis, Jostle with no imbalance and the former implementation of Party (green diamonds) show a similar behavior for the edge cut, the new implementation of Party (black circles) does compute less divergent (often identical) solutions here, reaching an average edge cut of 100.9. On the right hand side the average resource amount consumed by the different libraries is displayed. In case of the 100x100 grid, Jostle needs longest to compute its results, followed by Metis and Party, while Metis needs the largest amount of memory.

An important point that can be observed is that if the libraries were compared based on only the original graph, their rating would have been different. Metis (kMetis and pMetis) and Jostle with 1% and 3% imbalance allowance would have performed worse while Jostle with no imbalance and Party would have come off better, resulting in different conclusions.

The reason for the large variations is due to the libraries' implementation. During the refinement process, all vertices whose migration reduces the edge-cut by the same value are stored in an unsorted container. Vertices from the best bucket are chosen for migration, however, their order varies according to the vertex permutation.

The same observation can be made for partitioning libraries that randomize the vertex selection for the buckets. Using different seeds for the random number generator results in similar variations concerning the solution quality [Els02]. However, our approach is also applicable to implementations that are deterministic.

3.2 Implementation Enhancements

This section provides implementation details of the main components in the Party graph partitioning library. Though the new version is a complete rewrite, we focus on the realized improvements compared to the former code [Pre00]. First, we describe changes to the matching strategy that prevent the generation of star like graphs.

Then, we list the implementation of the Helpful-Set bisection refinement heuristic in detail.

3.2.1 Weight Limited Coarsening

The coarsening phase in Party is based on the Locally-Heaviest matching algorithm (LAM) that is presented in [Pre99]. This algorithm computes a 2-approximation of a maximum weighted matching in linear time. However, without modifications it will often create star-like graphs, meaning that some very heavy vertices with very high degree are generated. This makes the result unusable because neither is the structure of the coarsened graph similar to the original instance anymore nor is it possible to combine the vertices of a star-like graph appropriately such that a smaller graph of about half the size can be created for the next level.

To prevent this, we modify the LAM algorithm to only consider vertices as matching pairs if their combined weight does not exceed twice the weight of the lightest plus the weight of the heaviest vertex that occurs in the whole graph. This ensures that heavy vertices are only matched with light ones if at all, what leads to more equal weights. If too few matching pairs exist, a second round of the LAM algorithm is performed, what usually only happens in case of large variations of vertex weights. Additionally, we introduce a factor (set to 2/3) by that a graph has to shrink before the next level of the multi-level scheme is reached. This avoids running the expensive local refinement if a graph does not significantly differ in its size compared to its larger predecessor on the next higher level. If a graph does not meet this requirement, it is coarsened again.

3.2.2 The Improved Helpful-Set Implementation

The Helpful-Set heuristic is derived from theoretical observations as mentioned in chapter 2. However, although even if we know of the existence of a helpful and a balancing set, it is not clear how to find them quickly. Therefore, the helpful and balancing sets are built greedily, taking into account that some sets are overlooked.

The helpfulness of a single vertex is defined in a similar manner as the helpfulness of a vertex set and represents the edge cut reduction that would occur if a vertex changes its partition. To find the vertices with the highest helpfulness, they are stored inside a priority queue sorted according to their current value. Note that not all vertices but only those are included in the queue that have at least one neighbor in a foreign partition, what is called lazy-initialization [WC00]. Depending on the number of vertices and the edge weights, this queue is now either realized as a bucket, similar to other libraries, or alternatively as a red-black tree, reducing time and memory requirements if too many buckets were empty which often occurs in lower levels of the graph hierarchy.

Adaptive limitation The bisection refinement heuristic searches for l-helpful sets, that is a subset of nodes from either partition $V_1 = A$ or $V_2 = B$ decreasing the edge cut by l if moved to the other partition. If such a set is found, it is migrated and the algorithm then tries to find a balancing set in the same manner that eliminates the caused imbalance but also does not increase the edge cut by more than $l - 1$. If the search is successful, the set is moved, what results in a total edge cut reduction of at least 1. Otherwise the migration of the helpful set is undone and l is adjusted. The whole process is repeated until no more improvements can be made.

It has turned out that the crucial point when implementing the Helpful-Set algorithm is the choice of the helpfulness value l of the sets to be searched for. If l is chosen too low, then promising sets are overlooked, while setting l too high quickly increases the runtime but usually does not discover better sets. The former implementation in Party therefore uses a technique called *adaptive limitation* [DMP95] to determine l. Setting it to $cut/2$ initially, l is halved or set to the best occurring helpfulness if a l-helpful-set cannot be found in any of the two sets and doubled if the search has succeeded.

The improvements we made here can be characterized into several aspects. First, instead of using the same limit value l for both sets A and B, we introduce a separate one for each. This also leads to some changes in the adaptive limitation. Second, we include the possibility of having slight imbalances. This feature has also been implemented in Jostle and Metis, and it has proven to be quite successful since slight imbalances often make better edge cuts possible. If an imbalance of $i\%$, $0 \le i \le 100$ is required, a partition's weight may not exceed $\lceil |V|/2 * (100+i)/100 \rceil$. Especially in deeper levels of the multilevel approach it is not necessary (often not even possible) to completely balance the two sets. Thus, there is still some benefit even if fully balanced solutions are requested. Third, the former implicit balancing mechanism has been moved to the end of the algorithm and is performed explicitly, now always ensuring the requested bounds and increasing the reliability of the library. Figure 3.3 lists the new implementation.

```
01 HelpfulSet($A, B$)                       /* Current bisection is into partitions A and B */
02   $l_A \leftarrow l_B \leftarrow cut/2$                                   /* Initialize the limits */
03   while $l_A + l_B > 0$        /* Search for edge cut reductions until both limits are 0 */
04     if $l_A = 0$ or $2 \cdot l_A < l_B$                 /* Choose the more promising partition */
05       Swap($A, B$)                               /* Exchange names of the partitions */
06     $S_A$ = BuildHS($A, l_A, -d/2, s_{max}$)            /* Search for a $l_A$-helpful set $S_A$ */
07     if $h(S_A) < l_A$                 /* If the helpfulness of $S_A$ is smaller than wanted ...
08       $l_A \leftarrow b(S_A)$                            ... adjust the limit for the next search */
09       if $l_B > h(S_A)$    /* Test if a more helpful set can be found in the other partition */
10         $S_B$ = BuildHS($B, l_B, -d/2, s_{max}$)        /* Search for a $l_B$-helpful set $S_B$ */
11         if $h(S_B) \geq h(S_A)$              /* Name the partition with the better set '$A$' ...
12           Swap($A, B$)
13         else
14           $l_B \leftarrow b(S_B)$                 ... and reduce the limit of the other partition */
15       UndoBuild($S_B$)             /* undo the build operation for the less promising set */
16     if $h(S_A) < 0$                             /* Check, if a helpful set was found at all ...
17       $l_A \leftarrow b(S_A)$                                         ... and save its helpfulness */
18       continue                                  /* Proceed with the next loop in line 3 */
19     $l_A \leftarrow$min($l_A, h(S_A)$)                       /* Adjust the limit for the next search */
20     MoveSet($S_A$)                                                /* Move the helpful set */
21     $min \leftarrow (w(B) - w_{max}(B) - grace)^+$                    /* Determine the minimum ...
22     $max \leftarrow (w(B) - w_{min}(B) + grace)^+$   ... and maximum weight for the balancing set */
23     $S_B$ = BuildBS($B, 1 - h(S_A), min, max$)                     /* Search for a balancing set */
24     if $w_l \leq w(S_B) \leq w_h$ and $h(S_B) > -h(S_A)$        /* Check, if the balancing set is ok */
25       MoveSet($S_B$)                                     /* Yes: Move the balancing set */
26       $l_A \leftarrow l_A + \lceil \log l_A \rceil$                                   /* Increase the limits */
27       $l_B \leftarrow l_B + 1$
28     else
29       UndoBuild($S_B$)                            /* No: Undo the build operation and ...
30       UndoMove($S_A$)                                  the movement of the helpful set */
31       $l_A \leftarrow l_A/4$                                                  /* Reduce the limits */
32       $l_B \leftarrow l_B/2$
33   Balance($A, B, grace$)         /* Enforce balanced partitions by moving vertices greedily */
```

Figure 3.3: Implementation of the improved Helpful-Set bisection heuristic.

Like in the former implementation, the limits l_A and l_B are set to half the current edge cut initially (line 2). The following loop (line 3-32) is then performed until both limits are 0. If the currently first set's limit l_A is much smaller than B's one (line 4), the names of the sets A and B are exchanged (line 5). This ensures that a helpful set is first searched for in the more promising looking partition. During the search, the helpful set may neither become less than $-d/2$ helpful, where d is the average degree of a vertex, nor contain more than s_{max} (in our implementation 128) vertices (line 6). If a l_A-helpful set (S_A) (with weight $w(S_A)$ and helpfulness $h(S_A)$) cannot be found (line 7), the limit is reduced to the best helpfulness ($b(S_A)$) that has occurred during the search. Then, it is decided if a search in partition B is wanted (line 9). If so, S_B is determined (line 10). The set with the better helpfulness is named S_A (line 11,12) and kept while the other partition's limit is reduced and the set deleted (line 15). If the helpfulness of S_A is less than 0, the limits are adjusted (line 17,18) and the procedure is repeated, otherwise l_A is adjusted, S_A is moved to B (line 20) and the balancing phase is entered. There (line 21,22), the allowed weight interval $[min, max]$ of the balancing set is calculated. Due to the high vertex weights in lower levels of the multilevel method, this calculation includes a variable *grace* (that equals half the largest vertex weight), what introduces more flexibility. Within these bounds a balancing set S_B is then looked for that does not increase the edge cut by more than $1 - h(S_A)$ (line 23). If such a set can be found (line 24), it is moved to A (line 25) and the limits of both sets are increased (line 26,27). Otherwise, S_B is canceled (line 29), S_A is moved back to A (line 30) and limits l_A and l_B are decreased. Finally (line 33), if no more improvements can be made, the balance is checked and corrected if necessary by greedily moving the currently best vertices.

2-Processor SMP Multithreading To obtain more than two partitions, bisections can be applied recursively. Since their invocation describes a binary tree, a possible parallelization is straightforward. After processing the root node of the tree (that is bisecting the original graph), each of the two parts is then processed by one of the two processors. No communication is required between them, since the subproblems are completely independent, what makes the implementation very easy.

The question of how to use the second processor for the initial bisection remains. Note, that this root problem is the largest bisection problem of the whole computation and thus requires most of the resources. Nevertheless, in the current version of the Party library no further parallelism is implemented inside the bisection. Instead, we use the second processor to improve the partitionings' quality by computing the same problem (with different representations) twice and keeping the best result. To create an alternative representation, a graph permutation is required. Here, we apply the same mechanism as described in section 3.1.

3.3 Experimental Results

This section presents the results that we obtain applying the old and new implementation of Party and the state-of-the-art graph partitioning libraries Jostle and Metis. We first look at bisections computed by the two versions of Party, and then compare these results with those of Jostle and Metis. Next, we loosen the balance constraints to allow for a larger variation of the partition sizes before we conclude this section with the results of extensive tests that cover partitionings of two to 50 parts.

All experiments perform 100 runs with different graph representations according to our evaluation scheme and have been performed on a Pentium III 933 MHz dual processor system with 1 GB of main memory. Our test set contains well-known graphs that are listed in table 3.3. Since our main motivation comes from the field of FEM computations, most of the instances are originated in this area. We apply the evaluation method presented in section 3.1, but shorten the presentation to the average results.

3.3.1 Balanced Partitionings

First, we compare the improved implementation of the Helpful-Set method with the former one. While the old version of Party can only compute fully balanced partitionings, we run the new implementation with 0%, 1% and 3% imbalance allowance respectively. In these tests, we do not use multiple threads.

The average results are displayed in table 3.2. The numbers in this table show that in all tested cases the new implementation achieves a smaller average edge cut even if no imbalance is allowed. As expected, a higher imbalance allowance decreases the edge-cut in most cases, because the feasible solution set size is enlarged. However, in rare cases the opposite can sometimes be observed like for the 'dime.20' graph. Here, the average edge cut increases from 185.0 in case of no allowed imbalance to

Table 3.1: Graphs in the test set and some of their properties.

			degree				
Graph	$\lvert V\rvert$	$\lvert E\rvert$	min	avg	max	diameter	origin
144	144649	1074393	4	14.86	26	35	FEM 3D
4elt	15606	45878	3	5.88	10	67	FEM 2D
airfoil.1	4253	12289	3	5.78	9	43	FEM 2D
biplane.9	21701	42038	2	3.87	4	102	FEM 2D
crack	10240	30380	3	5.93	9	84	FEM 2D
dime.20	224843	336024	2	2.99	3	1105	FEM 2D dual
grid100	10000	19800	2	3.96	4	198	FEM 2D
m14b	214765	1679018	4	15.64	40	33	FEM 3D
memplus	17758	54196	1	6.10	573	10	digital memory circuit
ocean	143437	409593	1	5.71	6	229	FEM 3D dual
stufe.10	24010	46414	2	3.87	4	54	FEM 2D
t60k	60005	89440	2	2.98	3	495	FEM 2D dual
vibrobox	12328	165250	8	26.81	120	3	vibroacoustic matrix
wave	156317	1059331	3	13.55	44	38	FEM 3D

190.0 with 3% imbalance. This can be explained with the larger solution space, too, because there might be more and probably worse local optima now. A closer look on the detailed data of this experiment reveals that the cut size of the best solution found during all 100 runs is smaller for 3% imbalance than in the 1% or 0% case.

Comparing the computed edge-cuts of the old and new implementation, we report a quality improvement (QI) of 1.2% up to 19.6%. As table 3.2 shows, the improved version does neither require considerably more time nor memory. Hence, we claim the new version to be superior to the old one.

The results of Jostle and Metis are shown in table 3.3 and 3.4. Jostle accepts an imbalance parameter and we used values of 3% (default), 1% and 0% of allowed imbalance. The genuine version of Metis does not allow to adjust the imbalance parameter, instead it is set to constant 0% and 3% for the two versions pMetis and kMetis, respectively.

Comparing the resource requirements, Party always consumes least memory, but Metis (kMetis and pMetis) is fastest. In some cases, Party can compete with Metis, while in some others it is slightly slower than Jostle. This difference between the graphs is probably due to the applied matching strategies, resulting in a different number of graph levels and refinements.

Table 3.2: Average solution quality and resource requirements obtained by the old and new Party implementation. The best result is printed bold while results not within 10% (quality) and 25% (resources) of this value are shaded gray.

Graph	Party (old)	Party (new)					QI
	imbal.=0%	imbal.=0%	imbalance=1%		imbalance=3%		
	edge cut	edge cut	edge cut	pw_{max}	edge cut	pw_{max}	
144	6963.1	6730.2	**6682.9**	72844.6	6814.1	74152.6	3.5%
4elt	167.6	**148.6**	149.7	7854.9	149.4	7925.5	12.8%
airfoil.1	84.3	81.5	78.8	2139.7	**77.5**	2174.6	3.4%
biplane.9	90.5	79.9	78.7	10917.8	**77.4**	11043.7	13.2%
crack	203.3	192.5	192.2	5155.4	**191.3**	5221.1	5.6%
dime.20	213.4	**185.8**	189.5	113058.4	190.0	114377.9	14.8%
grid100	106.9	**100.2**	100.9	5042.8	100.9	5141.0	6.7%
memplus	5858.7	5791.4	5744.5	8965.9	**5642.3**	9139.8	1.2%
m14b	4127.0	**3956.8**	4004.0	108236.1	4081.8	110064.4	4.3%
ocean	551.1	496.1	505.9	72142.9	**391.6**	72932.5	11.1%
stufe.10	62.2	52.0	51.3	12073.2	**51.0**	12204.9	19.6%
t60k	100.3	91.9	88.7	30218.4	**79.4**	30836.2	9.1%
vibrobox	11974.8	11711.7	**11691.8**	6191.4	11730.0	6269.4	2.2%
wave	9445.8	9104.9	9070.8	78797.0	**9049.6**	80052.5	3.7%

Graph	Party (old)		Party (new)			
	imbalance=0%		imbal.=0%	imbal.=1%	imbal.=3%	all
	time	memory	time	time	time	memory
144	**1.949**	45741.8	2.000	2.200	2.318	**38744.7**
4elt	**0.054**	2008.9	0.060	0.061	0.061	**1855.9**
airfoil.1	**0.011**	537.1	0.014	0.014	0.014	**518.4**
biplane.9	**0.074**	**2186.9**	0.075	0.076	0.075	2494.1
crack	**0.035**	**1348.1**	0.046	0.046	0.046	1473.1
dime.20	**1.227**	**18678.6**	1.285	1.282	1.287	23215.9
grid100	**0.031**	**1027.0**	0.034	0.034	0.034	1144.2
memplus	**0.161**	**4620.6**	0.372	0.384	0.374	5774.8
m14b	**3.337**	71679.4	3.602	3.748	3.810	**61310.7**
ocean	**1.204**	**22517.8**	1.262	1.287	1.277	26266.7
stufe.10	**0.079**	**2435.8**	0.083	0.083	0.083	2780.9
t60k	0.241	**5152.4**	**0.234**	0.235	0.235	6325.5
vibrobox	**0.160**	7320.1	0.287	0.368	0.433	**5576.5**
wave	**2.051**	46399.3	2.234	2.388	2.471	**40790.6**

Table 3.3: Numerical results obtained with Jostle.

Graph	Jostle								
	imbalance=0%		imbalance=1%			imbalance=3%			all
	edge cut	time	edge cut	time	pw_{max}	edge cut	time	pw_{max}	memory
144	7047.8	2.257	7040.0	2.186	72765.6	7030.4	2.180	73263.4	47072.7
4elt	169.9	0.113	164.7	0.113	7836.7	162.8	0.115	7911.7	2721.4
airfoil.1	88.5	0.028	86.6	0.024	2134.4	84.0	0.024	2162.2	739.0
biplane.9	105.0	0.148	101.1	0.145	10903.9	99.4	0.145	10993.3	3338.8
crack	205.8	0.070	203.7	0.069	5137.0	201.4	0.071	5185.3	1794.5
dime.20	243.3	1.504	228.5	1.463	112942.7	224.3	1.462	113713.1	31531.6
grid100	117.0	0.062	111.9	0.065	5007.8	103.4	0.069	5038.1	1560.4
memplus	6183.7	0.356	6140.5	0.340	8966.5	6092.3	0.325	9085.6	5887.4
m14b	4221.7	3.355	4206.3	3.298	107916.2	4203.5	3.298	108592.4	72498.3
ocean	564.4	1.361	500.1	1.334	72319.8	438.6	1.330	73078.1	29339.7
stufe.10	64.8	0.151	62.8	0.148	12052.3	62.5	0.148	12118.0	3712.4
t60k	101.4	0.382	97.1	0.370	30129.8	92.8	0.369	30469.2	8577.2
vibrobox	11802.8	0.417	11858.3	0.392	6218.7	11821.1	0.402	6320.0	6800.3
wave	9404.2	2.392	9416.5	2.267	78628.5	9410.1	2.263	79262.9	48204.8

Table 3.4: Numerical results obtained with Metis.

Graph	pMetis			kMetis			
	imbalance=0%			imbalance=3%			
	edge cut	time	memory	edge cut	time	pw_{max}	memory
144	6855.9	1.321	73596.3	6989.6	1.515	72642.1	70080.5
4elt	160.5	0.060	3969.8	169.0	0.062	7888.2	3596.9
airfoil.1	84.7	0.016	1078.6	88.5	0.014	2168.9	982.5
biplane.9	84.8	0.081	4581.5	89.2	0.083	10949.4	4060.0
crack	199.8	0.043	2820.0	210.8	0.042	5188.9	2580.5
dime.20	192.0	1.011	41622.7	194.1	1.169	112598.3	36157.1
grid100	112.9	0.036	2147.2	126.4	0.035	5071.2	1910.7
memplus	6555.6	0.118	6739.4	6583.8	0.125	9126.8	6535.7
m14b	4056.9	2.163	114897.6	4145.0	2.452	107573.8	109672.5
ocean	504.5	0.837	40283.4	512.9	0.958	72108.5	36795.2
stufe.10	59.0	0.084	5084.7	61.7	0.090	12092.0	4507.3
t60k	96.5	0.225	11276.4	105.5	0.253	30192.2	9820.6
vibrobox	11792.4	0.158	10640.1	11857.4	0.148	6297.5	10419.6
wave	9381.0	1.372	74861.3	9441.4	1.558	78539.7	71069.1

To simplify the quality comparison, the computed edge-cuts are listed again for 0% and 3% imbalance allowance in table 3.5, which additionally states the edge-cut deviation. Note that we have extended pMetis with the imbalance feature, not the authors of this package.

One can see that Party computes the solutions with the lowest average edge-cut for almost all graphs of the test set, followed by Jostle and pMetis. Furthermore, it is evident that permuting the graphs' vertices has a large impact on the results computed by all libraries. The variation of the edge cut produced by Jostle is usually highest, followed by Metis, whereas Party usually calculates partitionings of more similar quality.

Summarizing the results, we have demonstrated that our new implementation of the Helpful-Set heuristic in the Party library is a powerful and reliable refinement tool that, applied inside a multilevel framework, can compete with and often even outperform state-of-the art graph partitioning libraries.

3.3.2 Less Restrictively Balanced Partitionings

In the following we increase the balance allowance and present the results for the 15% and 75% imbalance settings. It is obvious that by loosen the balance restrictions of the graph bisection problem, partitionings with fewer cut edges become possible. However, this decrease is often not continuous. Looking at the 100x100 grid for example, the optimal bisection width is 100. This is still the case if slight imbalances are allowed. Only if one partition may contain $49 \cdot 50 = 2450$ vertices or less (which corresponds to more than 50% imbalance), the edge cut can be decreased to 99 or less. Furthermore, local minima sometimes prevent the expected effects of increasing the imbalance value.

Our experiments show that the time and memory requirements equal those for the balanced setting. As before, Metis usually runs fastest. Party always consumes the smallest and Metis the largest amount of memory. Thus, we omit the exact numbers and focus on the partition quality. Table 3.6 contains the average edge-cut values obtained for the graphs of the test set.

One can see that a high imbalance allowance usually allows fewer cut edges. Again, there are a few exceptions to this rule. The most obvious instance is the 'ocean' graph. Although a good solution of 361.9 can be found with less than 3% imbalance, Party can only find an average of 499.6 cut edges with the less tight restrictions of

Table 3.5: The average edge-cut and its standard deviation during 100 runs computed by Jostle, Metis and Party with an imbalance allowance of 0% and 3% respectively. The best result is printed bold while results not within 10% of this value are shaded gray.

Graph	Jostle		pMetis		kMetis		Party (new)	
	edge cut	±	edge cut	±	edge cut	±	edge cut	±
no imbalance allowed								
144	7047.8	536.2	6853.2	210.3	7711.6	304.9	**6730.2**	77.0
4elt	169.9	27.2	161.9	23.8	217.6	26.6	**148.6**	14.5
airfoil.1	88.5	8.7	84.6	7.2	138.4	121.3	**81.5**	5.1
biplane.9	105.0	26.6	84.8	10.8	107.2	14.8	**79.9**	5.8
crack	205.8	15.5	200.1	13.9	263.3	15.6	**192.5**	7.3
dime.20	243.3	33.3	191.1	18.3	215.4	20.4	**185.8**	21.9
grid100	117.0	11.0	113.3	9.2	155.9	10.5	**100.2**	1.1
memplus	6183.7	208.5	6587.3	216.6	6802.6	191.3	**5791.4**	115.9
m14b	4221.7	569.9	4058.7	132.6	4552.8	408.6	**3956.8**	58.3
ocean	564.4	262.7	506.4	19.8	751.2	39.7	**496.1**	16.4
stufe.10	64.8	22.0	59.4	18.7	81.5	21.6	**52.0**	4.7
t60k	101.4	12.4	97.0	10.4	113.2	15.4	**91.9**	4.1
vibrobox	11802.8	380.7	11799.4	318.7	18279.4	592.2	**11711.7**	151.6
wave	9404.2	398.1	9372.1	346.0	10404.0	387.8	**9104.9**	192.1
allowed imbalance = 3%								
144	7030.4	541.6	6855.9	211.2	6989.6	236.1	**6814.1**	151.6
4elt	162.8	26.4	160.5	23.4	169.0	19.6	**149.4**	15.4
airfoil.1	84.0	8.3	84.7	7.0	88.5	8.3	**77.5**	5.0
biplane.9	99.4	26.2	84.8	10.8	89.2	11.7	**77.4**	5.0
crack	201.4	14.4	199.8	14.2	210.8	10.9	**191.3**	6.7
dime.20	224.3	29.5	192.0	20.3	194.1	18.2	**190.0**	24.6
grid100	103.4	7.7	112.9	9.9	126.4	7.9	**100.9**	0.6
memplus	6092.3	206.7	6555.6	223.2	6583.8	236.4	**5642.3**	114.9
m14b	4203.5	567.0	**4056.9**	131.4	4145.0	349.5	4081.8	124.0
ocean	438.6	296.2	504.5	21.5	512.9	49.8	**391.6**	64.6
stufe.10	62.5	22.0	59.0	16.9	61.7	18.5	**51.0**	5.6
t60k	92.8	14.1	96.5	9.8	105.5	49.9	**79.4**	4.9
vibrobox	11821.1	426.2	11792.4	319.1	11857.4	367.1	**11730.0**	207.9
wave	9410.1	393.7	9381.0	344.6	9441.4	352.7	**9049.6**	182.9

Table 3.6: The average edge-cut and its standard deviation during 100 runs computed by Jostle, Metis and Party with an imbalance allowance of 15% and 75% respectively. The best result is printed bold while results not within 10% of this value are shaded gray.

Graph	Jostle		pMetis		kMetis		Party (new)	
	edge cut	±	edge cut	±	edge cut	±	edge cut	±
allowed imbalance = 15%								
144	6990.1	557.6	**6850.8**	199.9	6991.2	248.5	6987.8	299.1
4elt	159.3	22.0	162.5	23.8	167.5	18.0	**150.8**	10.9
airfoil.1	72.2	13.4	84.6	8.4	81.5	10.9	**60.6**	5.9
biplane.9	94.0	22.8	84.6	10.1	89.4	12.5	**75.1**	6.7
crack	199.1	10.3	200.2	14.8	209.2	8.9	**189.5**	6.6
dime.20	210.7	25.8	195.0	21.6	193.4	17.1	**160.9**	11.3
grid100	**100.9**	4.3	112.7	9.4	126.1	7.9	101.0	0.8
memplus	5796.3	217.6	6569.2	208.5	6020.6	302.4	**5000.2**	97.4
m14b	4159.7	484.3	**4077.4**	223.2	4107.8	111.7	4186.8	151.0
ocean	422.4	290.4	504.0	20.7	503.2	54.9	**421.7**	49.6
stufe.10	58.3	22.6	62.0	21.1	60.6	19.0	**45.2**	9.6
t60k	78.5	17.3	95.3	14.7	89.8	17.0	**68.1**	7.0
vibrobox	**11681.7**	532.4	11800.6	326.5	11796.1	365.0	11695.6	234.7
wave	9410.1	475.3	9373.6	351.3	9456.3	348.6	**9363.0**	505.3
allowed imbalance = 75%								
144	5694.9	823.5	8049.1	1863.4	7710.1	1799.9	**5002.4**	310.1
4elt	133.9	17.5	167.1	23.4	165.4	24.6	**122.1**	9.9
airfoil.1	57.0	8.2	74.2	13.9	69.0	13.5	**54.5**	4.6
biplane.9	68.3	15.0	86.3	18.8	81.9	18.9	**56.6**	7.1
crack	153.3	25.3	187.5	20.3	190.2	19.6	**124.9**	6.4
dime.20	53.1	40.9	134.4	69.1	119.3	61.8	**28.7**	13.9
grid100	81.0	8.8	117.5	13.6	118.6	11.4	**74.2**	2.3
memplus	1960.2	298.5	3519.7	386.7	3600.0	346.0	**1561.0**	53.1
m14b	4063.7	628.6	4333.9	675.1	4316.3	1158.0	**3709.1**	607.5
ocean	**438.6**	188.2	520.3	109.2	513.9	67.5	499.6	84.3
stufe.10	56.9	17.8	62.0	18.8	59.3	16.7	**46.0**	4.9
t60k	68.3	6.4	80.6	15.7	77.4	13.7	**60.7**	6.1
vibrobox	8844.1	1026.1	12822.1	1134.8	14146.4	1425.5	**5551.0**	401.5
wave	6765.4	1851.9	11537.7	2427.5	10690.7	2017.8	**4894.8**	1167.2

75% imbalance allowance. Again, this can be explained with the strong increase of feasible solutions and the emerging of local minima, which makes the search in the refinement procedure much more complicated.

Nevertheless, apart from the 'ocean' graph, Party performs quite well and calculates partitionings that often outperform those found by Metis and Jostle. This is especially noticeable for very large imbalances. While for high imbalance settings the quality of the partitionings computed by Jostle is better than the one obtained using Metis, this differs for small imbalance allowances where Metis (either pMetis or kMetis) produces slightly better average results.

Concerning the deviation, one can see that variations occur for all three libraries, but again it is usually smallest for the Party library. However, from the results we obtain it is not clear if and how the reliability of the algorithms is influenced by weakening the balancing constraints, as in some cases the variation becomes smaller while the opposite can be observed for other graphs. We think that this might depend too much on the graphs' structure and hence no general rule can be stated.

3.3.3 Recursive vs. Direct Partitioning

So far we have only considered bisections. In the following experiments, we partition the graphs into two to 50 parts. We apply the evaluation scheme from section 3.1 and compare the average results of 100 runs. Besides our recursive bisectioning library Party we consider the recursive library pMetis and the direct k-partitioning library Jostle. While Metis and Jostle are sequential, we utilize one (1P) or two (2P) processors for Party, and denote the use of the additional first bisection with (2P+). No imbalance is allowed in all settings. For a better readability, we display all values relative to those obtained using pMetis and restrict the presentation to some selected examples that cover the observations of the whole range of results.

Figure 3.4 shows the results for the '144' graph. One can see that for this graph Jostle performs fastest, except if a bisection is requested. pMetis is next and Party needs longest to compute its solutions. With the one processor setting, Party is about a factor of 1.3 slower than pMetis, while the multi-threaded versions can reduce this gap due to the parallel speedup that becomes larger with an increasing number of partitions. The 2P+ setting needs much more time that the 2P one, what shows that generating a permuted graph is currently a very costly operation. Looking at the average solution quality, Party performs best for 10 partitions or less

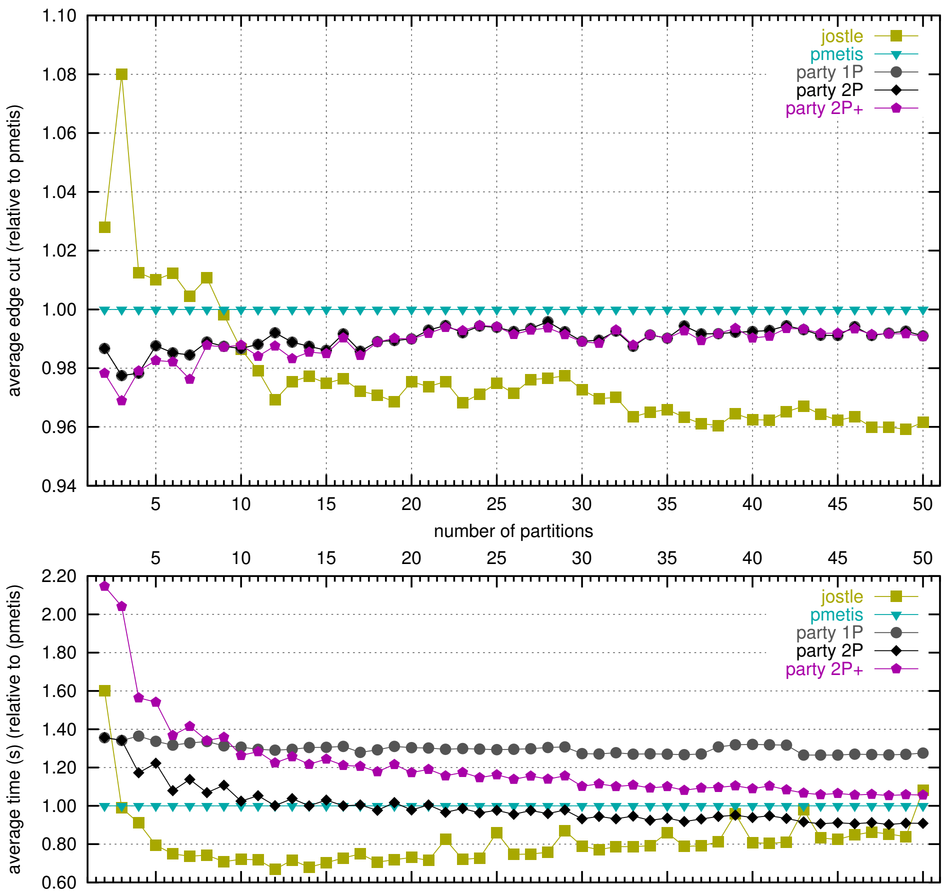


Figure 3.4: Average edge-cuts (top) and run-times (bottom) for the graph '144' for Jostle (yellow squares), pMetis (blue triangles), and Party (gray (1P)/black (2P) circles and magenta pentagons (2P+)).

and always better than pMetis. Note that the 1P and 2P setting always produce identical results. In case of more than 10 partitions, the direct approach in Jostle can outperform the recursive schemes of pMetis and Party by about 2-4%. While the 2P+ setting increases the run-time compared to the 2P one, which is most visible for a small number of partitions, the additional computations have only a small effect on the average partition quality.

The results obtained for the '144' graph reflect exactly what we expected before running the experiments. For a small number of partitions, the recursive approach is superior to the direct one while the opposite is true for a large number of partitions. pMetis performs faster than Party while the latter computes smaller edge cuts. Except for bisection problems where of course dedicated heuristics are the best

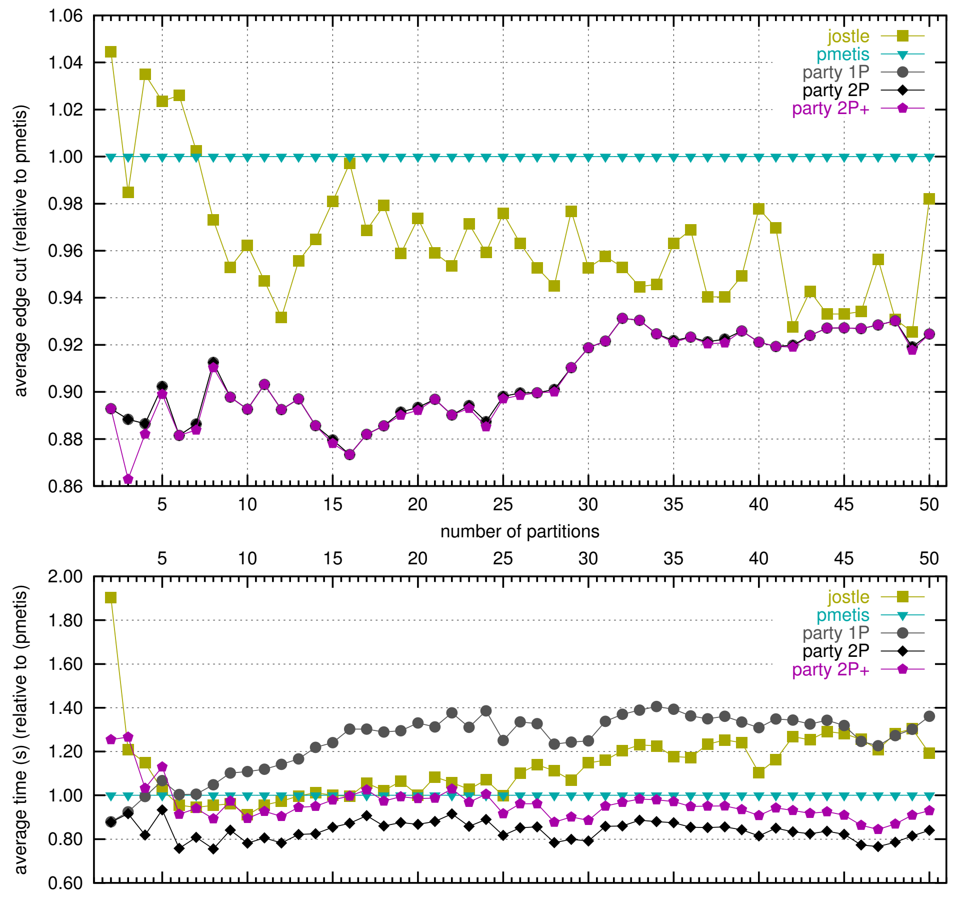


Figure 3.5: Average edge-cuts (top) and run-times (bottom) for the graph 'grid100' for Jostle (yellow squares), pMetis (blue triangles), and Party (gray (1P)/black (2P) circles and magenta pentagons (2P+)).

choice, direct approaches like Jostle are fastest. However, in the following we see that these conclusions are not transferable to other graphs in general.

On a regular grid, Party is known to perform very well. Hence, exceptionally good results can be obtained on this almost 4-regular graph as shown in figure 3.5. The bisection computed by Party contains less than 90% of the cut-edges as those calculated by Jostle or pMetis, and even for large partition numbers Party finds the best solutions. With 1 processor, our new implementation takes about 20% longer than pMetis and Jostle, while the threaded version is fastest.

In case of the 'biplane.9' graph, (figure 3.6), the single threaded version of Party is almost as fast as pMetis. The average solution quality computed by Party is about 5% better than the one obtained using pMetis. For a small number of partitions

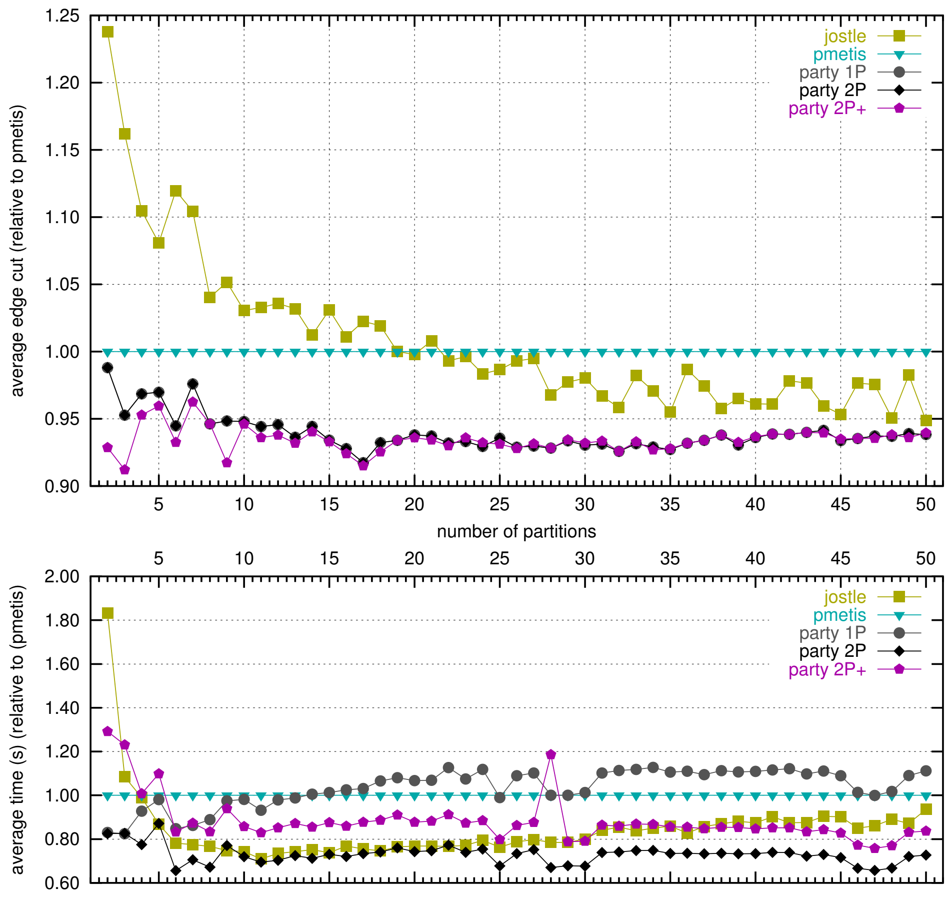


Figure 3.6: Average edge-cuts (top) and run-times (bottom) for the graph 'biplane.9' for Jostle (yellow squares), pMetis (blue triangles), and Party (gray (1P)/black (2P) circles and magenta pentagons (2P+)).

up to 20, both recursive approaches compute better results than Jostle, while from there on pMetis performs worst.

The values obtained when applying the libraries to the 'ocean' graph are given in Figure 3.7. Party computes the best average solutions while those of the direct approach in Jostle have a higher edge-cut than pMetis and Party, even for a large number of partitions. From 30 partitions on, the solution quality decreases even further, although the computation time enlarges.

The 'stufe.10' graph (Figure 3.8) is clearly a domain of recursive bisection heuristics due to its structure. Jostle is faster, but also computes solutions with a more than 50% larger edge-cut than Party. With a larger partition number, this difference becomes smaller, but also the run-time of Jostle increases.

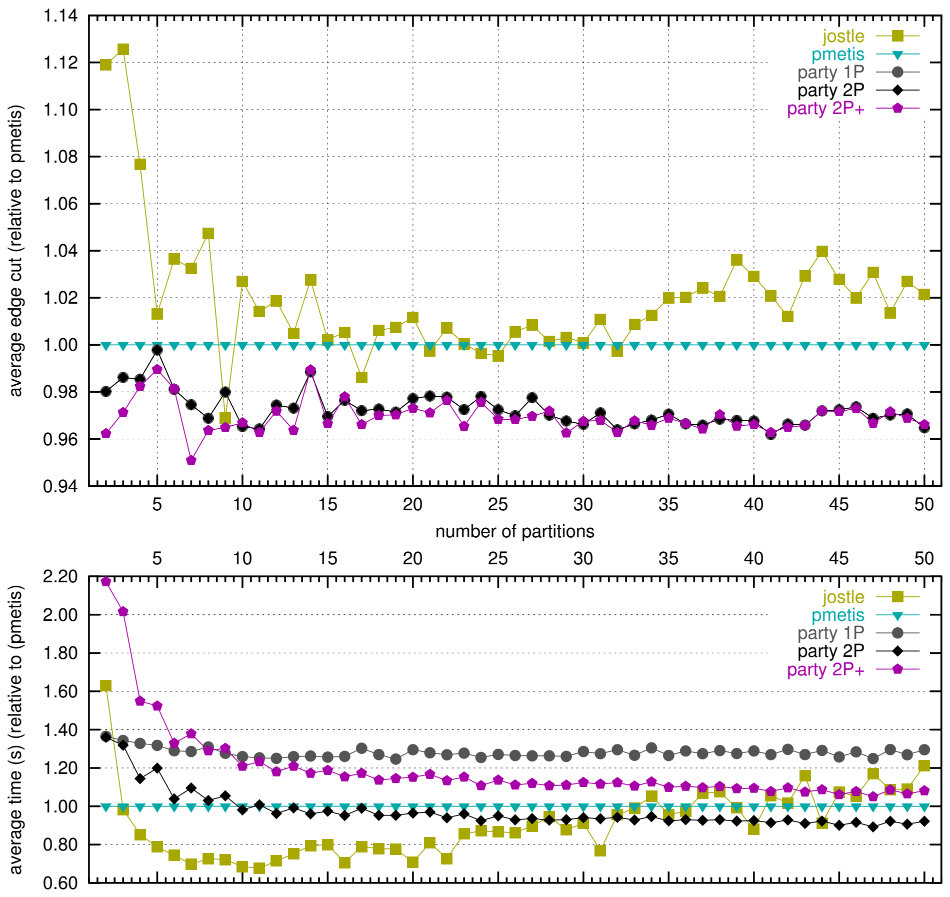


Figure 3.7: Average edge-cuts (top) and run-times (bottom) for the graph 'ocean' for Jostle (yellow squares), pMetis (blue triangles), and Party (gray (1P)/black (2P) circles and magenta pentagons (2P+)).

Summarizing our observations, we conclude that with the exception of a bisection a direct approach as implemented in Jostle (or kMetis) is usually faster than a recursive one. This is not surprising since the multilevel scheme has to be executed for each node of the recursive binary tree, while it is only performed once in a direct library. However, in many cases the run-time of Jostle increases faster with an increasing number of partitions than it does for the recursive approaches.

Concerning the average solution quality, the experiments show that the recursive libraries pMetis and Party compute much better results for small numbers of partitions than the direct approach in Jostle (and kMetis). Even for larger partition numbers, Jostle does not find significantly better solutions. This is somewhat unexpected since a direct approach has more choices to reduce the edge-cut.

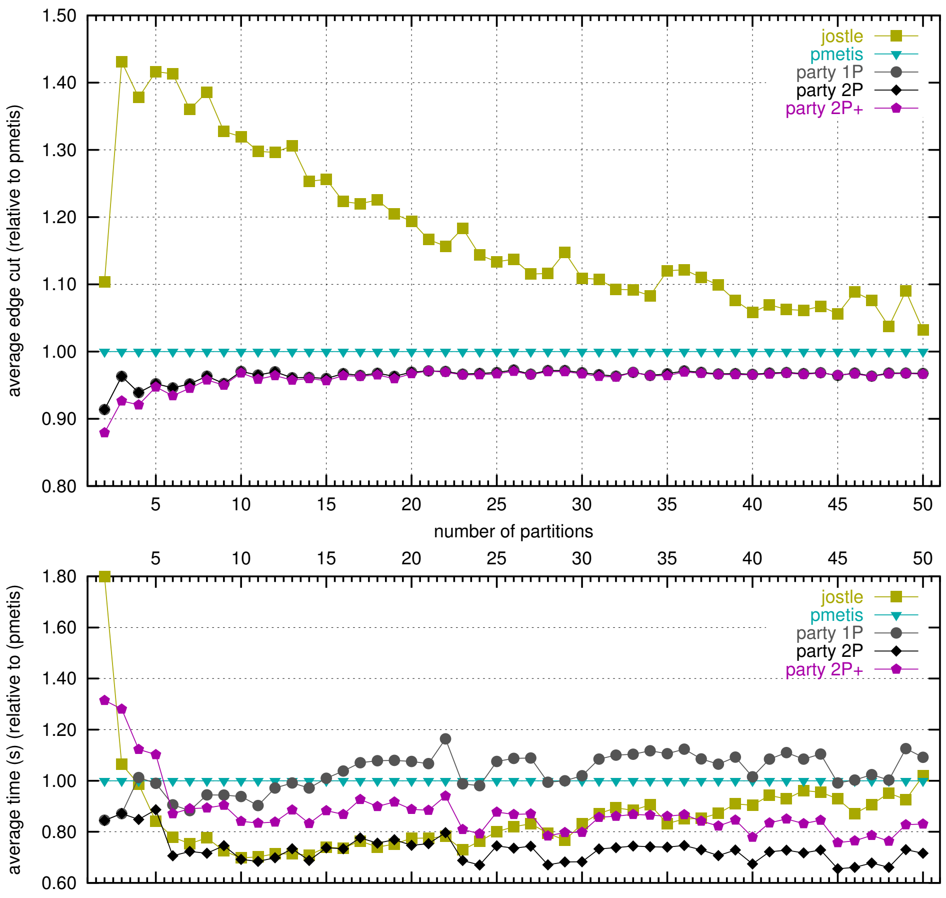


Figure 3.8: Average edge-cuts (top) and run-times (bottom) for the graph 'stufe.10' for Jostle (yellow squares), pMetis (blue triangles), and Party (gray (1P)/black (2P) circles and magenta pentagons (2P+)).

3.4 Upshot

The presented improved Helpful-Set heuristic is a powerful bisection refinement procedure. Together with the restricted matching strategy, our graph partitioning library Party combines two very effective key components for the multi-level approach.

The average solution quality that we obtain with the new implementation of Party often surpasses the results computed by Metis and Jostle, while the resource demands are comparable. The quality variation of the partitionings delivered by Party is smaller, which we ascribe to the new evaluation scheme that we applied during the development. Hence, we claim Party to be one of the leading graph partitioning libraries.

4 Diffusion

This chapter describes a technique to re-balance dynamically changing load in a network. It forms the basis for the key components of the algorithms presented in the next chapter. In contrast to the graph partitioning problem, the underlying model assumes that the distributed computations can be performed independently from each other, and does not consider data dependencies between the work packages. When determining a rearrangement of the calculations, the main attention lies on the traffic reduction in the network. Furthermore, the applied model usually assumes that the work load can be split arbitrarily. This problem is referred to as *dynamic load balancing*.

Formally, the load balancing problem is defined as follows. Given a connected graph $G = (V, E)$ representing the network with $n = |V|$ processing nodes where each node v contains work load w_v, the goal is to move load across the links $e \in E$ such that finally the weight of each node is (approximately) equal to

$$\overline{w} = \sum_{v \in V} w_v / n.$$

A flow $f \in \mathbb{R}^{|E|}$ is a *balancing flow*, if it satisfies the equation $\mathbf{A}f = w - \overline{w}$, where $\mathbf{A} \in \{-1, 0, 1\}^{|V| \times |E|}$ denotes the vertex-edge incidence matrix of G, and each column has exactly two entries -1 and 1 according to the two nodes incident to the corresponding edge. The signs implicitly define an arbitrary but fixed direction. In case of $f_e > 0$ we send load in the direction of the edge while $f_e < 0$ means that load is moved in the reverse direction.

Among all balancing flows, we are interested in the one that is minimal according to the $|| \cdot ||_2$-norm, that is the unique solution of the minimization problem

$$\min! \ \|f\|_2 = \sqrt{\sum_{e \in E} (f_e)^2} \quad \text{over all balancing flows } f.$$

If the global imbalance vector $w - \overline{w}$ is known, it is possible to find a solution to this problem by solving a linear system of equations [HB99]. But assuming a more restrictive environment allowing processors of the parallel network only to access information of their direct neighbors, load information has to be exchanged locally in iterations until a balancing flow has been computed.

Two subclasses of local iterative load balancing algorithms are the *diffusion* schemes [Cyb89, Boi90] and the *dimension exchange* schemes [Cyb89, XL97]. These two classes reflect different communication abilities of the network. Diffusion algorithms assume that a processor can send and receive messages to/from all of its neighbors simultaneously, while the dimension exchange approach is more restrictive and only allows a processor to communicate with one of its neighbors in each iteration. The *alternating direction* iterative scheme [EFMP99] represents a mixture of the diffusion and dimension exchange methods. It reduces the number of iteration steps needed for networks constructed by Cartesian products of graphs. The drawback of this scheme is that the resulting flow may have load migration loops tending to infinity.

4.1 The General Diffusion Scheme

The General Diffusion Scheme or First Order Scheme (FOS) [Cyb89, Boi90] is derived from Gauss-Seidel-Iterations and performs only local operations, i. e. data is only read from neighboring nodes.

Definition 1 (FOS). *Given a connected graph $G = (V, E)$, an initial work load distribution $w^{(0)}$ on its vertices and suitable constant α. In each iteration k, the First Order Scheme performs the operations:*

$$x_{e=(u,v)}^{(k-1)} = \alpha\,(w_u^{(k-1)} - w_v^{(k-1)})$$

$$f_e^{(k)} = f_e^{(k-1)} + x_e^{(k-1)}$$

$$w_v^{(k)} = w_v^{(k-1)} - \sum_{e=(v,*)\in E} x_e^{(k-1)},$$

with $x_e^{(k)}$ being the amount of load exchanged via edge e in iteration k and $f_e^{(k)}$ denoting the overall computed flow via edge e.

The First Order Scheme features two important properties. First, it balances the work load in a network [Cyb89], hence the load values $w^{(k)}$ converge to $\overline{w}$. Second, in the converged state the computed flow $f^{(\infty)}$ is the minimal balancing flow according to the $||\cdot||_2$-norm [DFM98, DFM99].

The Laplacian matrix $\mathbf{L}$ of G can be expressed as $\mathbf{L} = \mathbf{A}\mathbf{A}^T$. Defining $\mathbf{M} = \mathbf{I} - \alpha\mathbf{L}$, FOS can be written in matrix notation as

$$w^{(k+1)} = \mathbf{M}w^{(k)}.$$

$\mathbf{M}$ is called the *diffusion matrix*. It is nonnegative, symmetric, doubly stochastic, and of rank $n-1$. If we denote the eigenvalues of $\mathbf{L}$ in non decreasing order with λ_i, $1 \leq i \leq n$, it is known that $\lambda_1 = 0$ with eigenvector $(1, \ldots, 1)$ and $\lambda_n \leq 2D$, where D is the maximum degree of all vertices in G [CDS95]. If $\alpha < 1/\lambda_n$, $\mathbf{M}$ has the eigenvalues $\mu_i = 1 - \alpha\lambda_i$ in the range $-1 < \mu_n < \ldots < \mu_1 = 1$. The convergence rate of the First Order Scheme depends on $\gamma = \max\{|\mu_2|, |\mu_n|\} < 1$, the second largest eigenvalue of $\mathbf{M}$ according to absolute values, which is called the *diffusion norm* of $\mathbf{M}$. This rate expresses how fast the *error* $e^k = w^k - \overline{w}$ converges to zero. A system is called to be ϵ-balanced after the k^{th} iteration step if $\|e^k\|_2 \leq \epsilon \cdot \|e^0\|_2$.

The diffusion norm γ is influenced by the parameter α. A simple choice is to set all $\alpha_e = (1 + \text{maxdeg}(G))^{-1}$, but for many important graph classes better values are known as described in the next paragraph. The full knowledge of $\mathbf{L}$'s spectrum even allows to determine the optimal value [DFM99, EFMP99]. In this case, only m iteration steps are necessary where m is the number of different eigenvalues of $\mathbf{L}$.

This knowledge can be applied to reduce the communication in bus like interconnections. This network type allows each processor to communicate with every other, but only one transfer can be executed at a time. To reduce the bus allocations, a virtual topology is introduced which restricts the processors' communication to a few peers only. In order to speed up the load balancing process, topologies with a small degree and few different eigenvalues are of interest [EKM03].

4.2 Further Results on Diffusion

This subsection covers some extensions and generalizations of the First Order Scheme. First, we mention the Second Order Scheme which is obtained by overrelaxating FOS. The First Order Scheme is also applicable in inhomogeneous networks, which can contain vertex weights, edge weights or both. For selected network topologies, it is possible to compute the parameters for the optimal convergence speed. Furthermore, FOS is also applicable in dynamic networks.

4.2.1 Overrelaxation and Generalization

Similar to the overrelaxation known from the Jacobi-Iterations, this method can also be applied to the First Order Scheme. This results in the *Second Order Scheme* (SOS) [GMS98] of the form

$$w^{(1)} = \mathbf{M}w^{(0)}, w^{(k)} = \beta\mathbf{M}w^{(k-1)} + (1-\beta)w^{(k-2)},\ k = 2, 3, \ldots$$

The fastest convergence is achieved for $\beta = 2/(1+\sqrt{1-\mu_2^2})$, where μ_2 is the second smallest eigenvalue of the diffusion matrix $\mathbf{M}$. The Chebyshev method [DFM99] differs from SOS only by the fact that β depends on k according to

$$\beta_1 = 1,\ \beta_2 = \frac{2}{2-\mu_2^2},\ \beta_k = \frac{4}{4-\mu_2^2\beta_{k-1}},\ k = 3, 4, \ldots$$

Generalized, a *polynomial based* load balancing scheme is any scheme for which the work load $w^{(k)}$ in step k can be expressed in the form $w^{(k)} = p_k(M)w^{(0)}$ with $p_k \in \overline{\Pi}_k$, the set of all polynomials p of degree $\deg(p) \leq i$ satisfying $p(1) = 1$.

All polynomial diffusion schemes converge towards the $|| \cdot ||_2$-minimal balancing flow [DFM99]. If α and β are chosen optimally, SOS converges faster than FOS by almost a quadratic factor, while the Chebyshev method performs asymptotically identical to SOS.

4.2.2 Inhomogeneous Networks

To incorporate heterogeneous computing capacities of the processing nodes and nonuniform communication costs in the network, the diffusion schemes can be generalized. In such an environment, computations perform faster if the load is balanced proportionally to the nodes' computing speed s_v:

$$\overline{w}_v := s_v \cdot \left(\sum_{i=1}^{n} w_i / \sum_{i=1}^{n} s_i \right) .$$

Diffusion in networks with communication links of different capacities are analyzed in [DFM99, SG99]. It is shown that the existing balancing schemes can be modified, such that roughly speaking faster communication links get a higher load migration volume than slower ones. If the capacities of the links $e \in E$ are denoted with c_e, the minimal balancing flow computed by the diffusion schemes can be defined as

$$\min!\ \|f\|_2 = \sqrt{\sum_{e \in E} \frac{(f_e)^2}{c_e}} \quad \text{over all balancing flows } f.$$

These two generalizations can be combined [EMP02].

Optimal diffusion matrices

In order to improve the convergence of the First and Second Order Schemes, the edge weights of a graph can be adjusted such that the diffusion norm of its corresponding weighted diffusion matrix is minimized. In the following, we summarize the optimal values for some selected graph topologies [EMS02].

For unweighted edge-transitive graphs like Cycles, Hypercubes, complete graphs or the Star, the condition number is maximized if all edges are assigned the same weight due to the automorphisms. Hence, all edge weights can be set to 1.

Cayley graphs are constructed by generators Ω. In these cases, the condition number of the Laplacian matrix is maximized, if for any two edges e and f generated by the same generator $\omega \in \Omega$ the edge weights of e and f are equal.

For Cartesian products of two unweighted graphs G and H with $\lambda_2(G) \leq \lambda_2(H)$, the diffusion schemes on $G \times H$ can be improved by assigning the weight $\lambda_2(H)/\lambda_2(G)$ to the edges contained in G and 1 to the edges contained in H.

As a consequence, edges belonging to the same dimension of a Torus must have the same weight. Furthermore, a d-dimensional Torus can be viewed as a Cartesian product of d Cycles. Since the eigenvalues of a Cycle of length n are $2 - 2\cos(\frac{2\pi j}{n})$, $0 \leq j < n$, the polynomial based diffusion algorithms have their fastest convergence rate, if the edge-weights of cycle i, $1 \leq i \leq d$ are set to $(2 - 2\cos(\frac{2\pi}{n_1}))/(2 - 2\cos(\frac{2\pi}{n_i}))$, where $n_1 \leq n_2 \leq \cdots \leq n_d$ are the cycle lengths.

Other graphs with a similar structure are d-dimensional Grids. These are not Cayley graphs and it is known that edges of the same dimension do not necessarily need to have the same edge weight [DMN97]. However, considering them as Cartesian products of Paths of length $n_1 \leq n_2 \leq \cdots \leq n_d$, we can also improve the diffusion algorithms. Similar to the Torus, improved results are achieved by setting the edge weight of a dimension i to $(2 - 2\cos(\frac{\pi}{n_1}))/(2 - 2\cos(\frac{\pi}{n_i}))$.

The d-dimensional Cube Connected Cycles Network CCC(d) contains cycle and hypercube edges and is a Cayley graph. The optimal value of the condition number of the Laplacian of a weighted CCC(d) is achieved by setting the weights of all cycle edges to 1 and the weights of all hypercube edges to $a = 2 \cdot \sqrt{2}\frac{1}{\sqrt{d}} + \mathcal{O}(\frac{1}{d})$.

The d-dimensional Butterfly Network BF(d) contains path and cross edges, but is not a Cayley graph and therefore the optimal edge weights cannot be determined with the previously applied methods. It turns out that the condition number of the Laplacian is maximized if all edge weights are set to 1. However, extending the

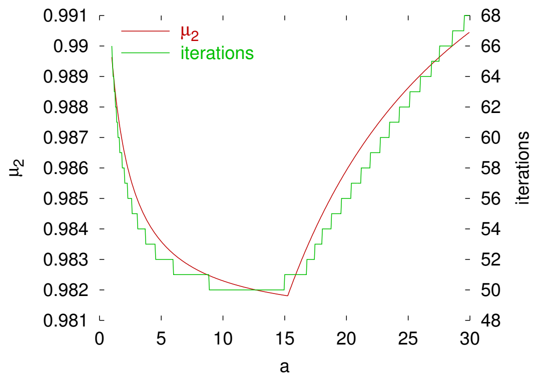


Figure 4.1: SOS SS $G(4 \times 16)$

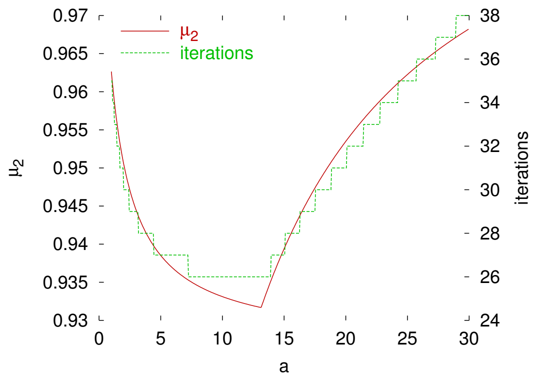


Figure 4.2: SOS SS $T(4 \times 16)$

$\mathrm{BF}(d)$ by wrap around edges our technique becomes applicable again. Similar to the CCC, the optimal condition number occurs when the weight of the cycle edges equals 1 and the weight of the cross edges are $a = \frac{1}{\sqrt{2}-1} + \mathcal{O}(1/\sqrt{d})$. However, there is no significant improvement of the condition number for large d. The same parameter a can be computed for the d-dimensional de Bruijn graph $\mathrm{DB}(d)$.

Experimental Results To show the effects of varying edge weights we have implemented a simulation program. While it is possible to determine eigenvalues of relatively small networks from the Laplacian itself, we are not able to do this for larger networks in a reasonable amount of time. Therefore, we determine the second smallest and largest eigenvalues of these graphs by either using explicit formulas or by reducing their calculations to the computation of eigenvalues of only parts of the original graph.

Prior to the first iteration of the simulation, the network's load is either distributed randomly (RS) over the network or placed onto a single node (SS), while we normalize the balanced load ($\overline{w} = 1$). The total amount of load is therefore equal to the total number of nodes n in the graph. We apply FOS and SOS and keep iterating until an almost evenly distributing flow is calculated. For our tests, we define this to be archived as soon as after the k^{th} iteration $||\, w^k - \overline{w}\, ||_2$ is less than 0.01. For both diffusion schemes, we have chosen the optimal value of $\alpha = 1/(\lambda_2 + \lambda_n)$, for SOS we set $\beta = 2/(1 + \sqrt{1 - \mu_2^2})$.

Figures 4.1 through 4.4 show some results of our experiments. In all experiments, we fix one edge weight to 1 (usually the cycle edges) while the other are weighted with a (usually the hypercube edges). For each selection of a on the x-axis the

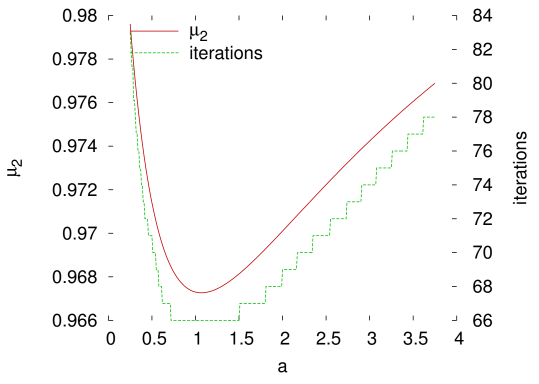


Figure 4.3: SOS SS CCC(8)

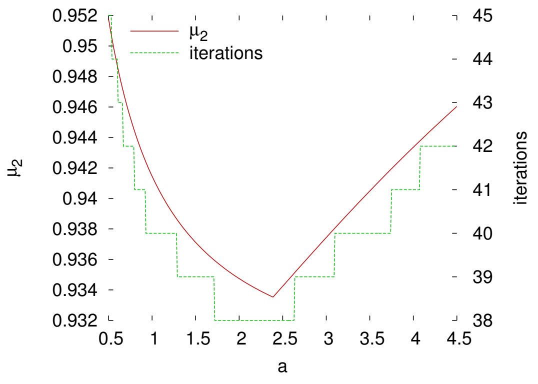


Figure 4.4: SOS SS DB(8)

resulting convergence rate μ_2 of FOS applied on the specific network type (left) and the number of iterations needed by SOS to compute a balancing flow (right) is shown. Note, that since the results are very similar for any combination of one of the schemes (FOS/SOS) and one of the initial load patterns (RS/SS), we have only included those for the SOS and SS.

As we can see from the figures, the closer a is to the optimal value a_{opt}, the smaller becomes the number of iterations needed to compute a balancing flow on all network types and savings up to 28% can be archived.

4.2.3 Dynamic Networks

The First Order Scheme can also be applied to balance load in dynamic networks where communication links fail from time to time or are present depending on the distance of nodes moving in space. Its convergence depends on the average value of the quotient of the second smallest eigenvalue of the Laplacian matrix and the maximum vertex degree of the networks occurring during the iterations [EMS04]. If $(G_k)_{k\geq 0}$ defines a sequence of graphs and $\Lambda_K = (\sum_{k=1}^{K} \lambda_2^k / d_{max}^k)/K$ is the average value of λ_2^k / d_{max}^k occurring during the first K iterations, the First Order Scheme needs at most K steps to ϵ-balance the system, where

$$K = \mathcal{O}\left(\frac{1}{\Lambda_K} \cdot \ln(1/\epsilon)\right).$$

If λ_2^k / d_{max}^k is always larger than some value $\frac{\lambda}{d}$, then at most

$$K = \mathcal{O}\left(\frac{d}{\lambda} \cdot \ln(1/\epsilon)\right)$$

iteration steps are required.

Experiments To test the load balancing in a dynamic environment we select a mobile ad-hoc network (MANET) model. The simulation area is the unit-square and 256 nodes are placed randomly within it. Prior to an iteration step of the general diffusion scheme, edges are created between nodes depending on their distance. Here, we apply the disc graph model (e.g. [WL02]) which simply uses a uniform communication radius for all nodes. After executing one diffusion step, all nodes move toward their randomly chosen way point, or stay there for a while before determining a new one. Once they have reached it, they pause for 3 iterations before continuing to their next randomly chosen destination. This model has been proposed in [JM96] and is widely applied in the ad-hoc network community to simulate movement. Note, that when determining neighbors as well as during the movement, the unit-square is considered to have wrap-around borders, that means nodes leaving on one side of the square will reappear at the proper position on the other side. Again, we average the results from 25 independent runs and the displayed charts are double logarithmic.

The *dynamic scheme* computes $\alpha_{(i,j)}$ for each edge (i, j) in every iteration, based only on local information: $\alpha_{(i,j)} = 1/\left(c \cdot \max\{\deg(i), \deg(j)\}\right)$. We compare it to three different static implementations. Each of them chooses a constant α to be used for all vertices in all iterations. The first one, 'clique', considers that possibly all nodes can become neighbors and therefore sets α to $1/(c \cdot |V| - 1)$. This choice guarantees convergence but for the price of a high number of required iterations, since the number of neighbors is usually much smaller and hence only a small amount of load is transferred. The algorithm 'first' tries to avoid this by investigating the initial configuration. It determines the highest degree of all nodes and sets α to $1/(c \cdot \text{maxdeg}(G_0))$. The disadvantages are that this operation requires some global view on the graph and the convergence is not ensured either. We should mention that the first iteration is not the one directly after the random placement of the nodes, but we always perform 25 rounds including movement only before the simulation starts to ensure similar graph properties in every iteration. A way to estimate the vertex degrees with less global knowledge is performed by 'probability'. Knowing the total number of nodes, the uniform communication radius r and the area of the simulation space, the expected number of neighbors is $|V - 1| \cdot r^2$. Hence, α is set to $1/(c \cdot |V - 1| \cdot r^2)$. Nevertheless, the resulting diffusion scheme might also diverge.

Figure 4.5 gives an example of a MANET and the load situation after 10 iterations using the dynamic approach. Since the communication radius is relatively small

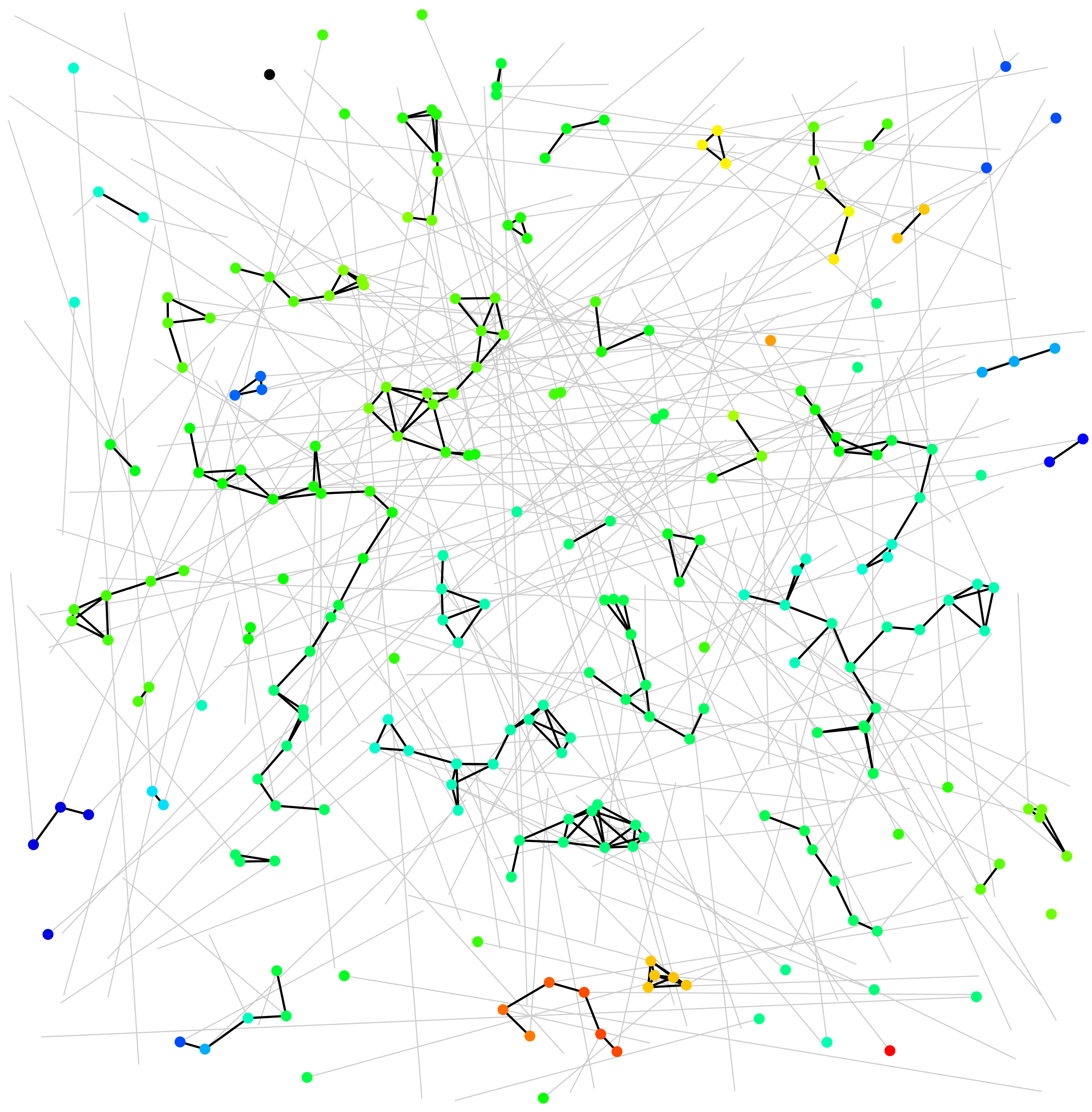


Figure 4.5: MANET: Load situation after 10 iterations. The simulation is run on the unit-square with wrap around. The initial load is placed on a single node. The disc radius is 0.1 and nodes move by a distance of 0.01 in each iteration. Nodes with a heavy load are displayed red while empty notes are drawn black. The thin grey lines indicate the movement direction.

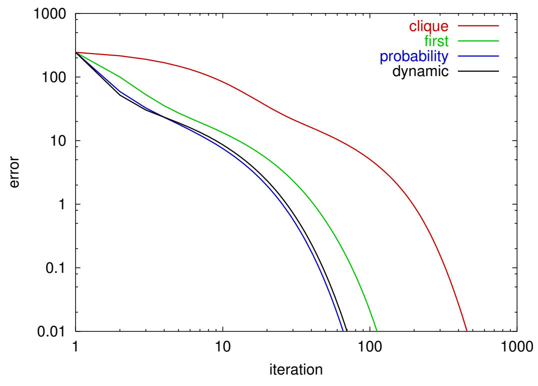


Figure 4.6: $r = 0.2$, $s \in [0.001 \ldots 0.001]$

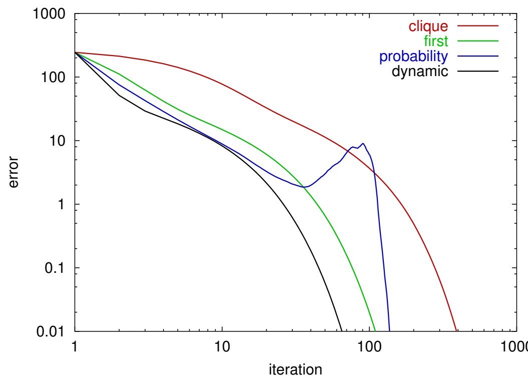


Figure 4.7: $r = 0.2$, $s \in [0.001 \ldots 0.005]$

(0.01), the graph is disconnected. One can see how the load has already spread into the network, indicated by the different colors. Vertices of the different connected components are almost balanced in this picture due to their slow movement speed.

Additional to the parameters SS/RS and c described we can alter the communication radius and the minimal and maximal vertex speed. However, we restrict our presentation to some selected results. In all experiments we place all load on a single vertex (SS) and set $c = 1.1$.

Figure 4.6 shows the error in every iteration of the general diffusion scheme for all four described approaches in a setting with a relatively large communication radius $r = 0.2$ and a slow movement rate $s = 0.001$. As expected, using 'clique' requires many iterations until the load is balanced. The 'first' approach performs already much better, but is again outperformed by the dynamic approach. A slightly faster convergence is archived by 'probability', which overtakes 'dynamic' after a few iterations. Figure 4.7 is based on a very similar setting. However, the increase of the maximal vertex speed causes convergence problems applying the 'probability' implementation.

By further increasing the vertex movement rate, the number of iterations decreases what is displayed in figures 4.8 and 4.9. These pictures are very similar to the one shown in figure 4.6, with the difference that through the higher speed of the vertices and the resulting faster changes in the graph structure speeds up the convergence rate of all test candidates. After a few iterations, the 'first' approach almost performs as fast as the 'probability' and the dynamic approach for speeds between $[0.1 \ldots 0.2]$.

The next two figures 4.10 and 4.11 show the results for a smaller communication radius. In both cases, using 'probability' does not lead to a balanced load situation

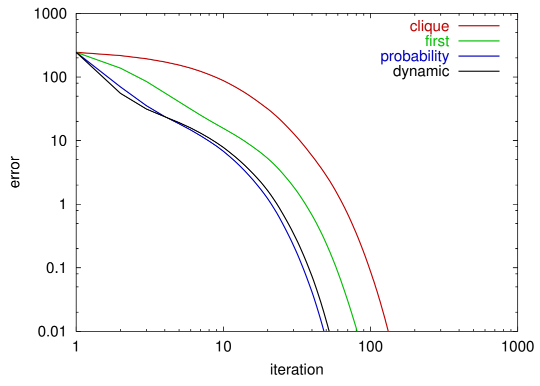


Figure 4.8: $r = 0.2$, $s \in [0.01 \ldots 0.05]$

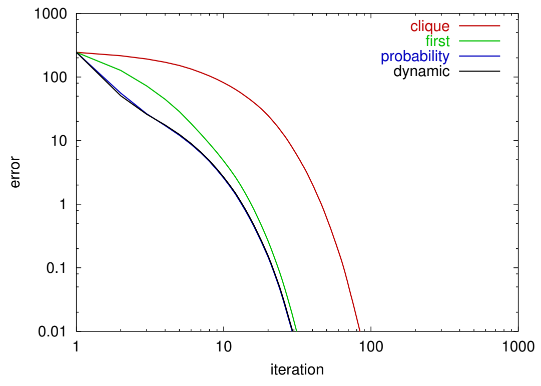


Figure 4.9: $r = 0.2$, $s \in [0.1 \ldots 0.2]$

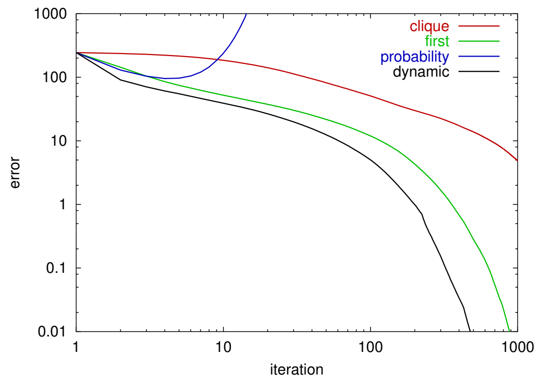


Figure 4.10: $r = 0.1$, $s \in [0.001 \ldots 0.001]$

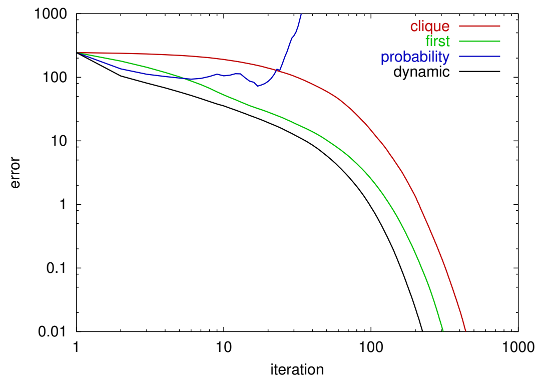


Figure 4.11: $r = 0.1$, $s \in [0.001 \ldots 0.05]$

since possibly more load is sent than available. The dynamic approach performs best, followed by 'first' and 'clique'. One interesting observation is, that accelerating the nodes highly affects the convergence rate of all algorithms. The reason for this is that now load is not only distributed over the edges but is also spread by the fast moving vertices that “visit” many other nodes.

This is even more the case in the following figures 4.12 and 4.13. In figure 4.12, the communication radius equals the movement rate of the nodes. The error applying 'first' decreases slower than for the dynamic approach, but at a certain point 'dynamic' is overtaken and 'first' ends up with the fastest computation. We are not sure of the cause for this behavior. Reasons could be found in the high movement rates. As mentioned earlier, convergence is not guaranteed for 'first' and maybe a very good constant has been determined for this setting. However, this means that the calculations performed for 'dynamic' are not optimal and leaves some open questions for further research.

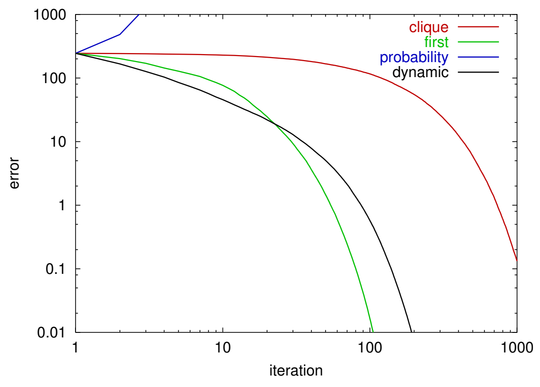


Figure 4.12: $r = 0.05$, $s \in [0.05 \ldots 0.05]$

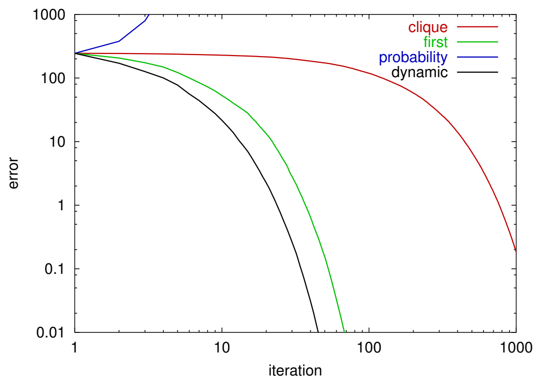


Figure 4.13: $r = 0.05$, $s \in [0.2 \ldots 0.2]$

The last figure 4.13 shows an example with an extremely high vertex speed set to 4 times the communication radius. Within this configuration load is primarily distributed by the movement of the vertices. Nevertheless, the dynamic scheme outperforms all others in contrast to the situation in figure 4.12.

Summarized, the dynamic scheme is a reliable load distribution strategy in dynamic networks and, compared to the three alternatives, performs fastest in most settings.

5 Shape Optimized Partitioning

This chapter introduces our novel graph partitioning and repartitioning heuristic Flux. In contrast to other libraries, Flux does not explicitly try to minimize the edge-cut, but implicitly focuses on good partition shapes. The shapes are optimized by diffusion processes that are embedded in a learning framework. Our experiments show that this method generates high quality solutions.

After giving a motivation, we introduce the learning framework and describe the evolution process of the new diffusion scheme FOS/C. We reveal some of FOS/C's properties and propose a number of enhancements for a fast and reliable implementation. Next, we present experimental results with our sequential and parallel library, where we do not only consider the established metric edge-cut but also alternative and more suited metrics like the number of boundary vertices.

5.1 Motivation

The existing graph partitioning heuristics provide good solutions and are quite fast. However, although great progress has been made in this area, many questions remain [Hen98]. While the global edge-cut is the classical metric that most graph partitioners optimize, it is not necessarily the metric that models the real costs of an application. In FEM computations for example, the true communication volume can significantly differ from the number of cut edges. In this case, the number of vertices situated at partition boundaries reflects the amount of information to be exchanged much more accurately. Another questionable point is the commonly applied norm. In synchronized computations, the slowest processor specifies the overall speed, hence the maximum norm is appropriate, while the usually applied edge-cut is a summation norm.

Dynamically changing applications often require a work load distribution that guarantees both, a low overhead caused by the load migration and little communication during the calculation. This results in a multi-objective optimization, com-

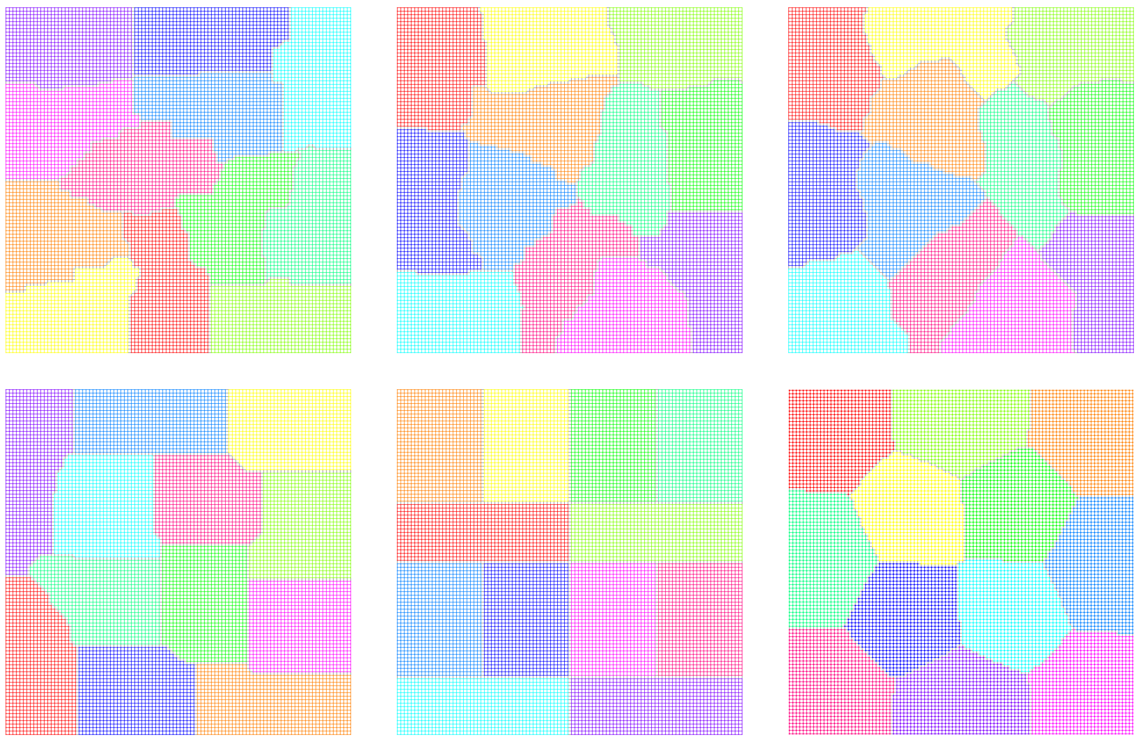


Figure 5.1: Dividing a grid into 12 partitions using the software libraries pMetis, kMetis, vMetis, Jostle, Party and the new shape optimizing approach (top left to bottom right).

prising the dynamic load balancing as well as the graph partitioning problem. An unbalanced partition π_t has to be transformed into a balanced distribution π_{t+1} while obeying the two objectives.

Most of the existing implementations first determine how much load to migrate and then restrict the exchange steps of the local refinement process according to this number. Better solutions can be obtained by integrating the migration costs directly into the objective function of the improvement procedure [BO98, SKK00, DHB02].

Our new heuristic proceeds this way, while it focuses on optimizing the partition shapes rather than the edge-cut. It is based on the same framework as an earlier approach [DPSW00] which iteratively decreases the partition aspect ratios. However, the latter algorithm has a couple of drawbacks concerning balancing and parallelization. We overcome these limitations by replacing the framework's most important steps by diffusive operations.

This combination results in a heuristic that delivers solutions which already differ from those of other libraries on the first view due to the round looking domains. For example, figure 5.1 displays the partitionings of different implementations for a regular grid. As one can see, the shape optimizing approach (bottom right) delivers partition shapes that are close to a circle where the graph structure allows this.

Although we cannot prove any quality bounds of our new heuristic yet, we conclude from our experiments that the solutions we obtain often outperform existing heuristics according not only to the shape and the number of boundary vertices but surprisingly also to the edge-cut. Furthermore, the proposed approach is also applicable to repartition a graph and keeps the number of migrating vertices small. Last but not least, the new heuristic contains a high degree of parallelism.

5.2 The Bubble Framework

The Bubble Framework [DPSW00] has evolved from simple greedy algorithms computing bisections of graphs. It is related to clustering algorithms [Mac67], but in contrast to graphs the latter operate on points in n-dimensional space and their implicitly given Euclidean distances. Starting with an initial, often randomly chosen vertex (seed) per partition, all sub-domains are grown simultaneously in a breadth-first manner. Colliding parts form a common border and keep on growing along this border – "just like soap bubbles". After the whole graph has been covered and all vertices of the graph have been assigned this way, each component computes its new center that acts as the seed in the next iteration. This is usually repeated until a stable state, where the movement of all seeds is small enough, is reached. This procedure is based on the observation that within "perfect" bubbles, the center and the seed vertex coincide. Distances in this framework may either be chosen as Euclidean distances or as path lengths in a graph. In the latter case, no geometrical information is required. Summarized, a bubble algorithm mainly consists of the following three phases that are also illustrated in figure 5.2:

Init A vertex for each partition is determined. These vertices act as the seeds in the first iteration.

Grow Starting from their seeds, all partitions grow in a breadth-first manner until all vertices are assigned.

Move All partitions determine a new center vertex. These vertices become the seeds in the next iteration.

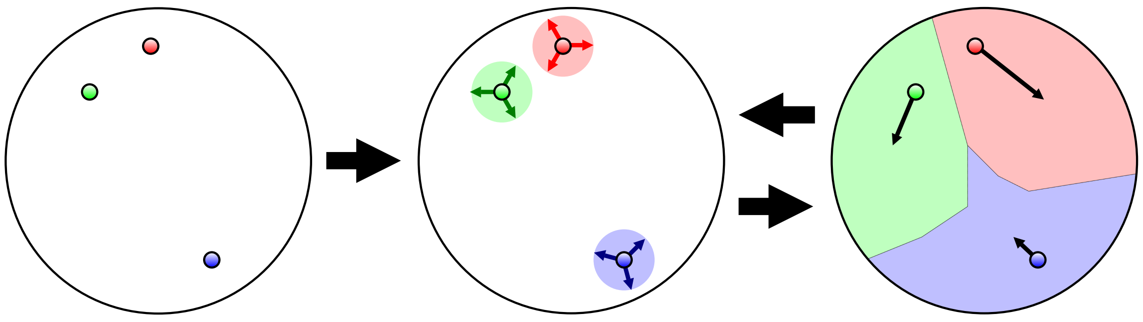


Figure 5.2: The three operations of the learning bubble framework: Init: Determination of initial seeds for each partition (left). Grow: Growing around the seeds (middle). Move: Movement of the seeds to the partition centers (right).

These three operations can be implemented in several ways. We first summarize existing implementations before introducing the new diffusion based method.

5.2.1 Previous Implementations

To our knowledge and apart from simple greedy heuristics there are two implementations that apply the bubble framework to solve FEM graph partitioning problems. The first one is part of a former version of the Party graph partitioning library. There, the implementation of the three phases can roughly be described as follows:

Init The initial seeds are determined randomly.

Grow Starting from every seed a breadth-first algorithm on the graph is applied. During this process the partitions alternately acquire one of their free neighbor vertices until all vertices are assigned.

Move The vertex with the minimal maximal distance to all other vertices of the same partition becomes the new seed.

This approach shows several problems. The initial placement of the partitions may be very bad, requiring many iterations until it is fixed, but even then the partition sizes usually vary extremely. The time spent on finding new seeds is quite large since a breadth-first-search has to be solved for every vertex. Moreover, the partition quality is not considered at all. Another important disadvantage is that the growing phase cannot be parallelized because vertices are assigned serially and earlier assignments have a great impact on later decisions.

A second approach is implemented in a former version of the FEM simulation tool PadFEM [DPSW00]:

Init The first initial seed is randomly chosen among the vertices with smallest degree. Then, to determine the seed for the next partition, a breadth-search is performed with all seeds as starting points. The last vertex found becomes the seed for the next partition. This is repeated until all seeds have been determined.

Grow The smallest partition with at least one adjacent unassigned vertex grabs the vertex with the smallest Euclidean distance to its seed.

Move The new seed of a partition becomes the vertex for which the sum of Euclidean distances to all other vertices of the same partition is minimal. To find this vertex quickly, some successive approximation is used.

This algorithm solves some of the problems we have seen in the first approach. The initial seeds are distributed more evenly over the graph. Since the smallest possible partition gathers the next vertex, more attention is paid on the balance, and also the determination of the center has been improved to work faster. By including coordinates in the choice of the next vertex, the partitions are usually also geometrically well shaped (and connected), which is the main goal of this approach. Other quality metrics are not considered. By relying on vertex coordinates this approach is only applicable if these are provided, and sometimes the Euclidean distance does not coincide at all with the path length between vertices. This can often be observed if an FEM mesh contains “holes”, in which case a partition may be placed around it. It is a general problem when working with coordinates and occurs more heavily for example in space-filling-curve based partitionings [SW03]. Experiments also reveal that the selection mechanism, though improved by preferring under-weighted partitions, does still not lead to sufficiently well balanced domains [DPSW00]. Hence, to fix this, some additional computations are added after the last bubble iteration. Concerning a possible parallelization, the situation stays the same as described before because the selection process of the vertices is still strictly serial.

5.3 Diffusion Based Mechanisms

This section describes how we integrate diffusion into the Bubble Framework. The main idea is based on the observation that load primarily diffuses into densely connected regions of the graph rather than into sparsely connected ones. Following this observation, we expect to identify sets of vertices that possess a high number of internal and a small number of external edges.

To assign the vertices to the partitions, we execute the diffusion algorithm exactly P times, each time with a different kind of load. To distinguish between these loads we color them with colors from 1 to P, respectively. After all load is distributed, we place the vertices to that partition they obtained load from. If a vertex contains more than one kind of load, meaning that it could be part of more than one domain, it is assigned to the partition of which it received the highest amount from. Hence, a partition can crowd others out of parts of the graph if it itself already contains a higher load nearby. The implementation of the operations Grow and Move can be described as follows:

Grow Independently for each partition, load is placed on the center vertices and distributed applying a diffusion scheme. Each vertex is assigned to that partition it has received the highest load amount from.

Move The vertex containing most load of color p becomes the new center vertex of partition p.

Performing these two steps and looking at the properties of a diffusive load distribution, we realize that the vertex with the highest load amount is the previous center vertex again. Hence, the centers and therefore the partitions do not move and no learning occurs. To overcome this problem, we introduce an alternative initial load placement for the diffusion process. Instead of only placing load on the single center vertex, we distribute it evenly among all vertices of a partition. Doing so, we obtain three different combinations of initial load placements and resulting vertex assignments:

Grow/Assignment Place load on the center vertex, perform the diffusion process and assign the vertices according to the highest load amount to obtain a partitioning.

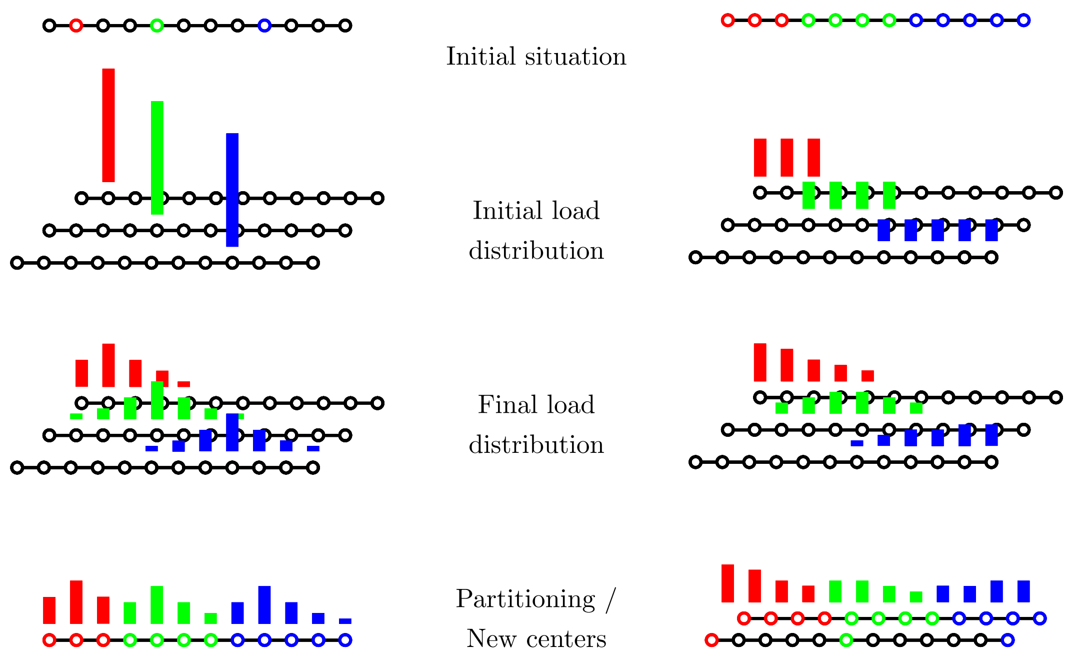


Figure 5.3: Schematic view: Placing load on single vertices (left) or a partition (right), the diffusion process and the assignment of the vertices to the partitions (left and right) or determination of the centers (right) according to the load.

Grow/Consolidation Place load evenly distributed on all vertices of the partition, perform the diffusion process, and assign the vertices according to the highest load amount to obtain a partitioning.

Move/Contraction Place load evenly distributed on all vertices of the partition, perform the diffusion process, and for each partition select the vertex with the highest load as the new center.

The three combinations are also illustrated in figure 5.3. On the left hand side, load is placed on the center vertices and the diffusion process leads to a partitioning, while placing load on the partitions can either be used to improve a partitioning or to determine new center vertices as shown on the right. Note, that we denote the vertices we place load on as set of source vertices. Depending on the operation, this set contains either the single center vertex or all vertices of a partition.

The operations Grow/Assignment and Move/Contraction are required for the Bubble Framework, Grow/Consolidation is optional and can be applied in between, even several times. Our experiments show that multiple invocations often improve the solution quality.

The total load amount W that we place in the system is set to the number of vertices $|V|$ of the graph. Note, that the choice of W only scales the final load distribution and therefore has no real impact on the solution, as long as it is the same value for all partitions. When performing a Grow/Consolidation or Move/Contraction operation, we evenly distribute that load on all source vertices. Hence, vertices of smaller partitions will contain more load to be spread into the graph, which supports balancing. Adjusting the load amount per partition is a way to balance the partition sizes and is discussed later.

For the Init phase, several alternatives are possible. As mentioned, on the one hand a random distribution may place vertices suboptimal, but on the other hand it is more likely that vertices in dense regions are chosen. In case of a repartitioning, it is possible to perform a Contraction to obtain centers. This can also be done after vertices have been assigned to a partition randomly. Currently, the following choices are implemented in our library:

Init/Random Center Randomly determine P single disjoint vertices as partition centers.

Init/Repartition Perform a Move/Contraction on a given partitioning to obtain the center vertices.

Init/Random Partition Compute a balanced random partitioning and proceed with Init/ Repartition.

Inserting the proposed diffusive operations into the Bubble Framework, the resulting algorithm roughly looks like as sketched in figure 5.4. The input consists of the graph G that is also capable of storing load and flow vectors, and the parameters i and l specifying the number of the different iterations to be performed.

The outer iteration (line 01) starts determining single center vertices for each of the partitions. If no partitioning π is present, it either selects random vertices (line 03) or computes a random vertex assignment (line 05). Now, in case of an existing partitioning, the center vertices are then obtained by performing a Move/Contraction, consisting of the a load distribution (line 09), the diffusion process (line 10) and the center determination (line 11).

The following inner loop (line 12), which has to be executed at least once, contains the Grow/Assignment and Grow/Consolidation operation. Note, that for each partition the set of source vertices S contains either the single center vertex or all the

```
00  Algorithm DiffusiveBubble(G, i, l)
01      in each iteration i
02          if not $\pi$ exists
03              case random-center
04                  c = determine-random-centers(G)
05              case random-partition
06                  $\pi$ = random-partitioning(G)
07          if $\pi$ exists
08              parallel for each partition p
09                  $w_p$ = distribute-load-on-parts(G, p, $\pi$)
10                  $w_p$ = diffusion(G, p, $w_p$)
11              c = determine-centers(G, w)
12          in each loop l
13              parallel for each partition p
14                  $w_p$ = distribute-load-on-sources(G, p, S)
15                  $w_p$ = diffusion(G, p, $w_p$)
16              $\pi$ = determine-partitioning(G, w)
17      return $\pi$
```

Figure 5.4: Sketch of the Diffusive Bubble algorithm.

partition's vertices, depending on the previous operation. The load w_p is distributed evenly among the source vertices (line 14), distributed via the diffusion process (line 15), and finally the vertices are assigned to obtain a partitioning π (line 16).

The sketched algorithm mainly contains a collection of loops. Of those, the loops over the partitions (lines 08 and 13) and the diffusion operation can be fully parallelized. Also the vertex assignment is a distributed operation, only the maximum computations during the contraction requires a global view on the whole partition. Another interesting point is that the presented algorithm does not contain any explicit objectives. These are somewhat hidden inside the diffusion processes of the growth and movement operations.

The most important and most costly part of the algorithm is the calculation of the diffusion. To be applicable, the solution of the applied scheme must possess two properties. As already mentioned, it is necessary for the learning framework that load should spread faster into dense regions of the graph. The diffusion schemes from

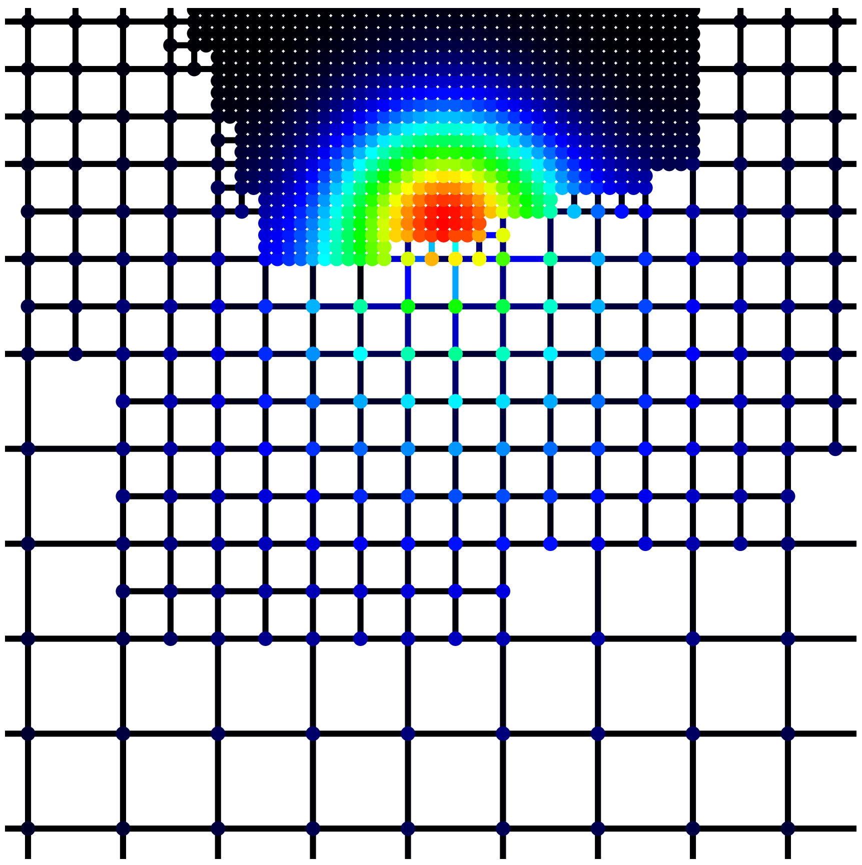


Figure 5.5: Load distribution originated at a single seed after 50 FOS iterations. Vertices with high load are colored red while empty vertices are black.

chapter 4 fulfill this constraint because they are known to compute a $||\cdot||_2$-minimal flow. However, to obtain connected sub-domains and to reduce the migration during a repartitioning, the load must be distributed in a hill-like manner with its peak close to the originating center. Figure 5.5 gives an example of such a distribution. Note, that although the vertices in the lower part are closer to the seed vertex, much more load reaches the vertices inside the denser upper part than those in the lower one since there are more paths into this region.

The diffusion schemes presented in chapter 4 balance the load completely. Since this contradicts the requested hill-like load distribution, they are not directly applicable and have to be modified. Three possible ways are described and discussed in the following. In an evolutionary process, we have first looked into the two approaches FOS/L and FOS/A. However, both of them are superseded by FOS/C, and we therefore mainly focus on this scheme.

5.3.1 The Limited First Order Scheme

A first idea to obtain a hill-like load distribution is to stop FOS before it converges. This means that we perform some FOS iterations before we interrupt the scheme and define the current load distribution as the final result. We refer to this approach as Limited First Order Scheme (FOS/L).

The combination of FOS/L and the Bubble Framework computes promising looking partitionings when comparing them with those of other heuristics. As an example, figure 5.6 shows the average results of a 16-partitioning obtained on the 'biplane.9' graph. The Bubble Framework is run with 4 learning iterations and 4 loops, respectively. The other libraries included in this experiment are those described in chapter 2, and the bars display the minimum, maximum and average results from 100 runs together with the standard deviation as mentioned in chapter 3.

Besides the edge-cut and the balance, the number of external edges, boundary vertices, and the resulting amount of information to be send and received by a partition is displayed, all stated in maximum, Euclidean and summation norm, respectively [1]. One can see that the edge-cut (and hence the number of external edges) computed by the shape optimizing approach is comparable to the results of Metis and Jostle. However, the number of boundary vertices (and hence the send/receive volume) of the new approach is significantly smaller in all norms. Similar results are obtained for other test instances [Sch04].

Although applying FOS/L yields good results already, especially when considering that the Bubble Framework is a global approach, one major problem remains: Determining the number of FOS iterations until the diffusion process is interrupted. This number is difficult to choose, since it depends on the graph, the number of partitions as well as on the current partition placement. If too few steps are performed, the load has not spread widely enough and vertices may remain completely empty, while too many iterations equalize the load such that it is no longer possible to determine a good partitioning. Since we are unable to state a general rule to obtain a good number, we have selected some appropriate values by hand for the experiments, often determined in a large number of runs. Of course, this is usually not applicable in practice. Thus, although FOS/L serves as a working proof of concept, a more reliable diffusion process is required.

[1] A summary of applied metrics and norms is given in section 5.7.1

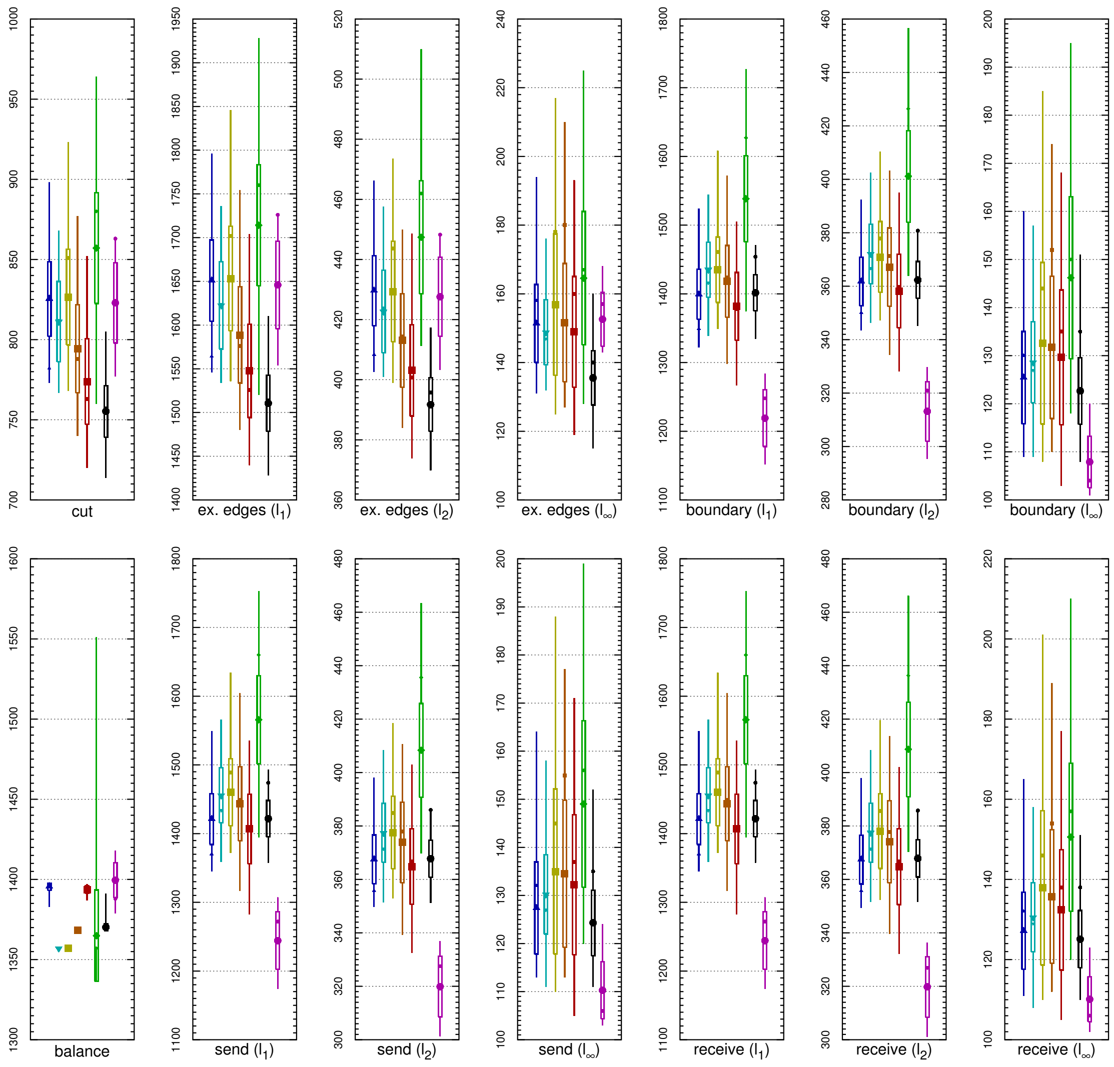


Figure 5.6: Detailed results obtained for the 12 partitioning of the 'biplane.9' graph. Results are shown (from left to right) for: kMetis (dark blue triangle), pMetis (light blue triangles), Jostle (squares) with 0% (yellow), 1% (orange) and 3% (red) imbalance allowance, Party (old) (green diamonds), Party (new) (black circles) and the shape optimizing approach applying the limited diffusion scheme (magenta circles). Each bar displays the average value of 100 independent runs with a large mark, the standard deviation of the values with a wide bar, the minimum and maximum values with thin bars and the result for the first run with a small mark.

5.3.2 The First Order Scheme with Absolute Draining

In this subsection we introduce the FOS/A diffusion scheme. In contrast to the original First Order Scheme, this scheme does not converge towards the completely balanced load situation. Instead, it is disturbed such that the converged solution has similar properties as the load distribution after some FOS iterations described in the last subsection. This eliminates the problem of finding a suitable number of FOS/L iterations since one can always iterate until convergence.

The disturbance is realized by decreasing the load on each vertex by an absolute value δ_A. Note, that if a vertex contains less load than δ_A, only the existing load is subtracted, and therefore all load values remain non-negative. To keep the total load amount in the system constant, all subtracted load is added equally distributed to the set of source vertices $S \subseteq V$. As before, for each partition this set contains either the single center vertex or all the vertices from that partition. Formally, the proposed scheme can be described as follows.

Definition 2 (FOS/A). *Let $G = (V, E)$ be a connected graph and $S \subset V$ be the chosen set of source vertices. The entries of the vector s are either set to $s_v = 1/|S|$, if $v \in S$, or 0 otherwise. Let $\delta_A > 0$ be the absolute drain constant and $W > 0$ the total load in the system. Furthermore, set $\alpha_{(u,v)} = 1/(1 + \max\{deg(u), deg(v)\})$ for each edge $e = (u, v)$ in G. In iteration k, let $w_v^{(k)}$ denote the load on vertex v and $x_e^{(k)}$ the flow exchanged over edge e. Initially, set $w^{(0)} = 0$ and $\Delta^{(0)} = W$. Then, the iteration scheme* FOS/A *performs in each iteration i the following computations:*

$$\begin{aligned}
x_{e=(u,v)}^{(k)} &= \alpha_e \cdot \left(w_u^{(k)} - w_v^{(k)}\right)\\
t_v^{(k+1)} &= w_v^{(k)} - \sum_{e=(v,*)} x_e^{(k)} + s_v \Delta^{(k)}\\
d_v^{(k+1)} &= \min\left(t_v^{(k+1)}, \delta_A\right)\\
w_v^{(k+1)} &= t_v^{(k+1)} - d_v^{(k+1)}\\
\Delta^{(k+1)} &= \sum_{v \in V} d_v^{(k+1)}
\end{aligned}$$

The first two steps of the scheme are almost equal to those of FOS, extended by the addition of the subtracted load Δ. The last three steps decrease the load and calculate the total subtracted load for the next iteration.

During the first iteration, all load is placed on the source vertices. Then, it diffuses into the graph, similar to the original First Order Scheme, because the subtracted load is relatively small compared to the amount on a vertex. The load continues to spread, more and more vertices are reached, and therefore Δ increases. At some point, however, the furthest vertices only obtain less than δ_A load, which is completely subtracted and sent back to the sources. Hence, nothing remains on these vertices, the spreading slows down and eventually stops.

Figures 5.7 through 5.10 show some important variables of a sample experiment on the 'biplane.9' graph with a single source vertex. In figure 5.7 and 5.8 we can see that during the first iterations of FOS/A, the amount of subtracted load Δ increases quickly, but the increase slows down such that the change $|\Delta^{(k)} - \Delta^{(k-1)}|$ becomes finally zero. Consequently, the exchange $\sum_{e \in E} x_e$ over the edges has reached a stable state and therefore also the load l_v on each vertex does not change any more.

However, the convergence is not smooth. Figures 5.9 and figure 5.10 report the load on the vertices that have at least δ_A load, called inner vertices, and those with less than δ_A load which are denoted as border vertices. One can see that the number of inner vertices mainly grows, but not continuously. Especially in later iterations the set of inner vertices sometimes even decreases again. Hence, also the load on the border vertices oscillates. At some point, however, there are no more changes.

The displayed data is collected in an experiment on an unstructured graph, what allows the behaviour to be explained as follows. In the beginning, the load on the vertices is much higher than δ_A, hence it spreads from the seed vertex into the graph in all directions. At some point, it might reach an area with a different structure, hence the distribution speed changes. For example, if the vertex degree in an area is larger than the average, the load flows faster. This causes load that has been sent into another direction to be reverted and vertices, that have already received more than δ_A load before might become empty again. Hence, there occur wave-like effects in the distribution until every reachable part of the graph is 'known' by the diffusion process and the load has been placed accordingly.

Although all our experiments show that the FOS/A scheme converges, we have been unable to prove this yet. Finding the iteration from which on the sets of inner and border vertices become stable turns out to be an important unsolved question due to the involved minimum computation in the diffusion scheme.

Nevertheless, we have implemented FOS/A into our framework and run numerous tests [Sch05]. As an example, some frames of a repartitioning experiment with the 'slowtric' benchmark[2] are displayed in figure 5.11. Here, a mesh sequence of 101

[2]The properties of this benchmark are described in section 5.7.2.

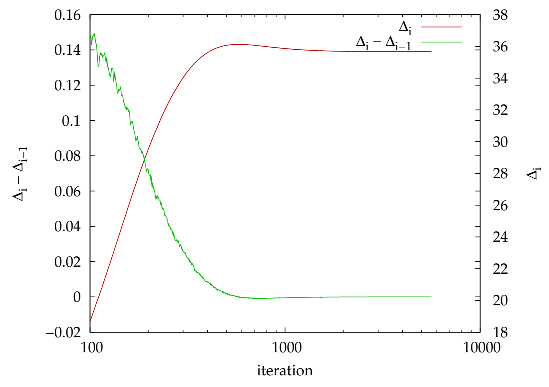


Figure 5.7: Δ^i and its changes.

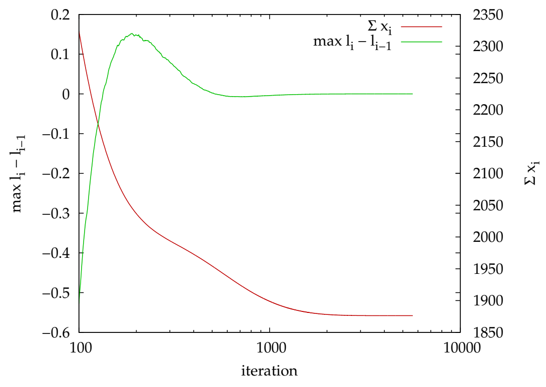


Figure 5.8: Total flow and load changes.

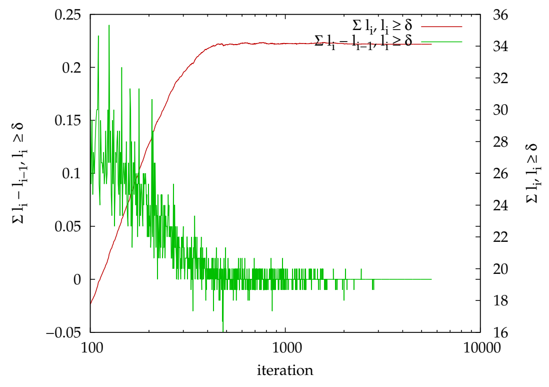


Figure 5.9: Inner load and its changes.

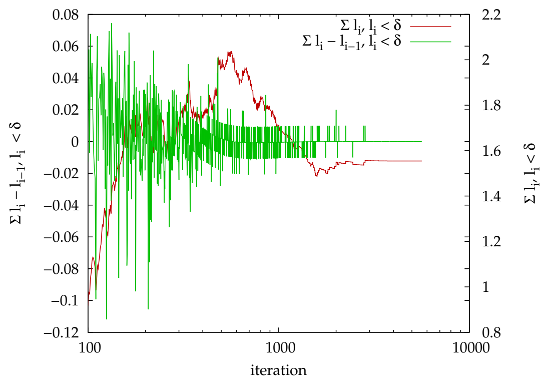


Figure 5.10: Border load and its changes.

frames is generated according to three circles rotating around the center of the squared simulation area. The circle sizes differ and due to a similar attractor placed in each of the circles, the area around the smaller circles is refined more deeply. Furthermore, the overall number of triangles increases slightly over time. Besides Metis and Jostle, we run the shape optimizing approach with randomly chosen initial centers, $W = |V|$ load in each partition, a drain value of $\delta_A = 0.01$, and perform 1 iteration and 12 loops to partition and repartition the meshes into 12 sub-domains.

The solutions of Metis are shown in the middle left column. While the initial solution looks acceptable, there seem to be some problems in later balancing steps. A closer look to e.g. frame 50 reveals that one partition is degenerated into three parts, one of them only consisting of a few vertices. Metis applies a local exchange heuristic that usually takes care of a few isolated vertices and assigns them to adjacent partitions. However, other frames show even single isolated triangles, and we guess that this problem is related to the parallelization of the exchange proce-

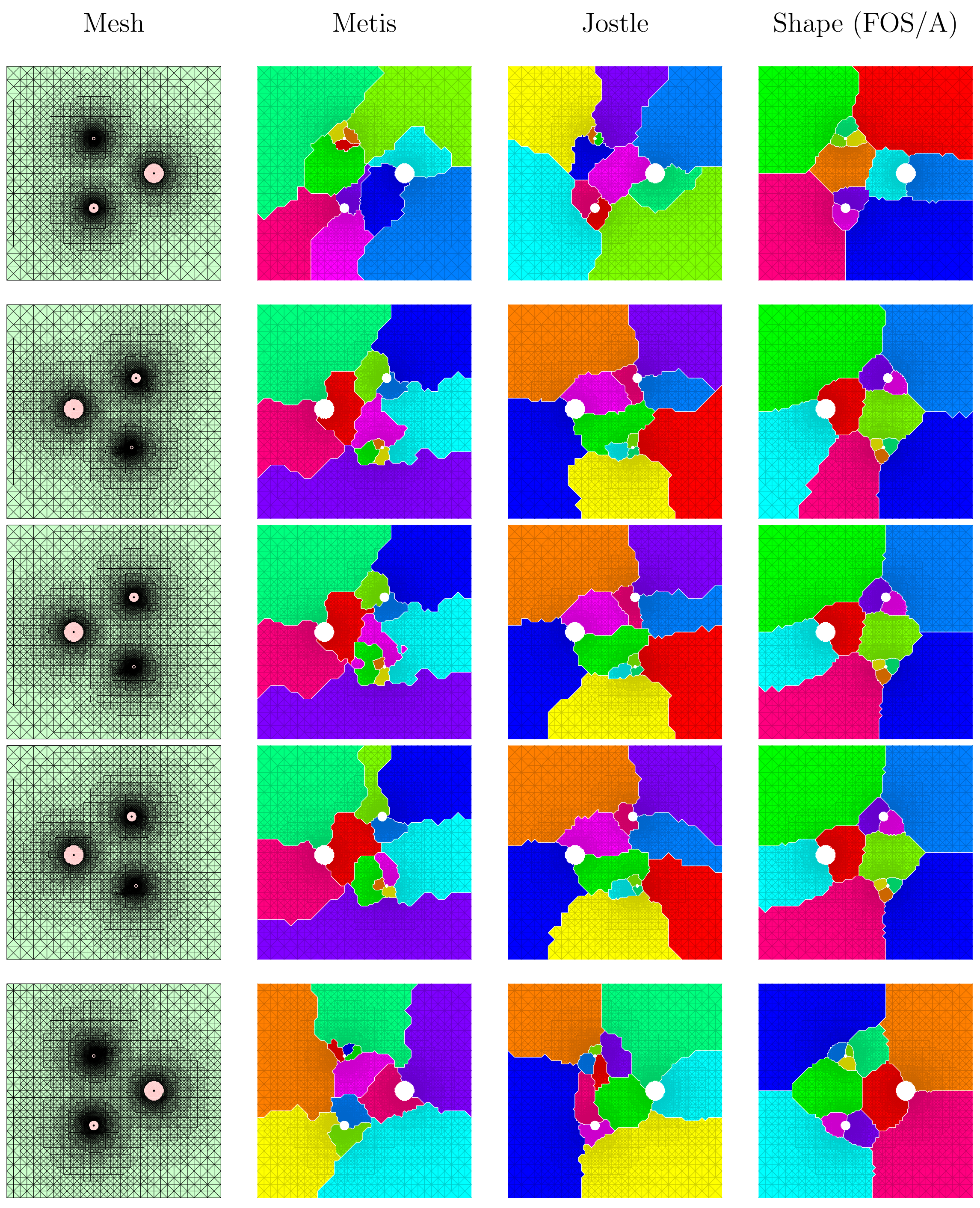


Figure 5.11: Frames 0, 49, 50, 51, and 100 of the 'slowtric' benchmark. From left to right, the FE mesh and the solutions computed by Metis, Jostle and the shape optimizing approach applying FOS/A are shown.

dure. Furthermore, partitions sometimes are thin or contain long extensions into their neighbors, which deteriorates the partition shapes and increases the boundary length and the communication volumes. The distributions calculated with Jostle (middle right) are usually of a better shape, though partitions are occasionally disconnected. When looking at the partition movement, it is interesting to see that domains in deeply refined areas (e.g. around the smallest circle) try to follow these locations. This property is even more distinct in the solutions obtained with the new approach (right). Furthermore, the shape optimized approach computes straighter boundaries and also computes connected domains.

The recorded metrics are displayed in figure 5.12. The left column contains the data according to the $||\cdot||_1$-norm while on the right side the $||\cdot||_\infty$-norm has been applied. The $||\cdot||_2$-norm is omitted. The first row contains the balance, displayed as the maximal size of a partition. It reveals the imbalance setting of Metis to 3%, although sometimes up to 6% are reached. The parallel version of Jostle is invoked with the same value, but ignores this setting. Our scheme does not provide an imbalance parameter, but usually achieves partition sizes with less than 5% imbalance with some exceptions in other benchmarks.

The next four rows contain the edge-cut, the number of boundary vertices, the communication volume (send and receive volumes are added) and the partition diameter. The first three of them are related due to the small vertex degree of the graphs. One can see that the values of the solutions computed by Metis are higher than those for the partitionings obtained with Jostle or the shape optimizing approach. This general observation holds for all of our benchmark sets. Looking at the diameter, Metis computes disconnected domains in some frames. In other benchmarks we can observe this even more frequently. The same holds for Jostle, though degenerated partitions occur less often.

The last row displays the migration (outgoing and incoming elements are summed up). Here, Metis seems to follow a different strategy than the two other libraries, moving either very little or a huge amount of data while the migration volume of Jostle and Party/DB is more constant over the frames.

Concerning the run-time, our sequential implementation of the shape optimized approach applying FOS/A requires about two to three magnitudes longer than the libraries Metis and Jostle. The disturbance of FOS/A slows down the diffusion process even more and its multiple invocation in the learning framework requires lots of computing resources.

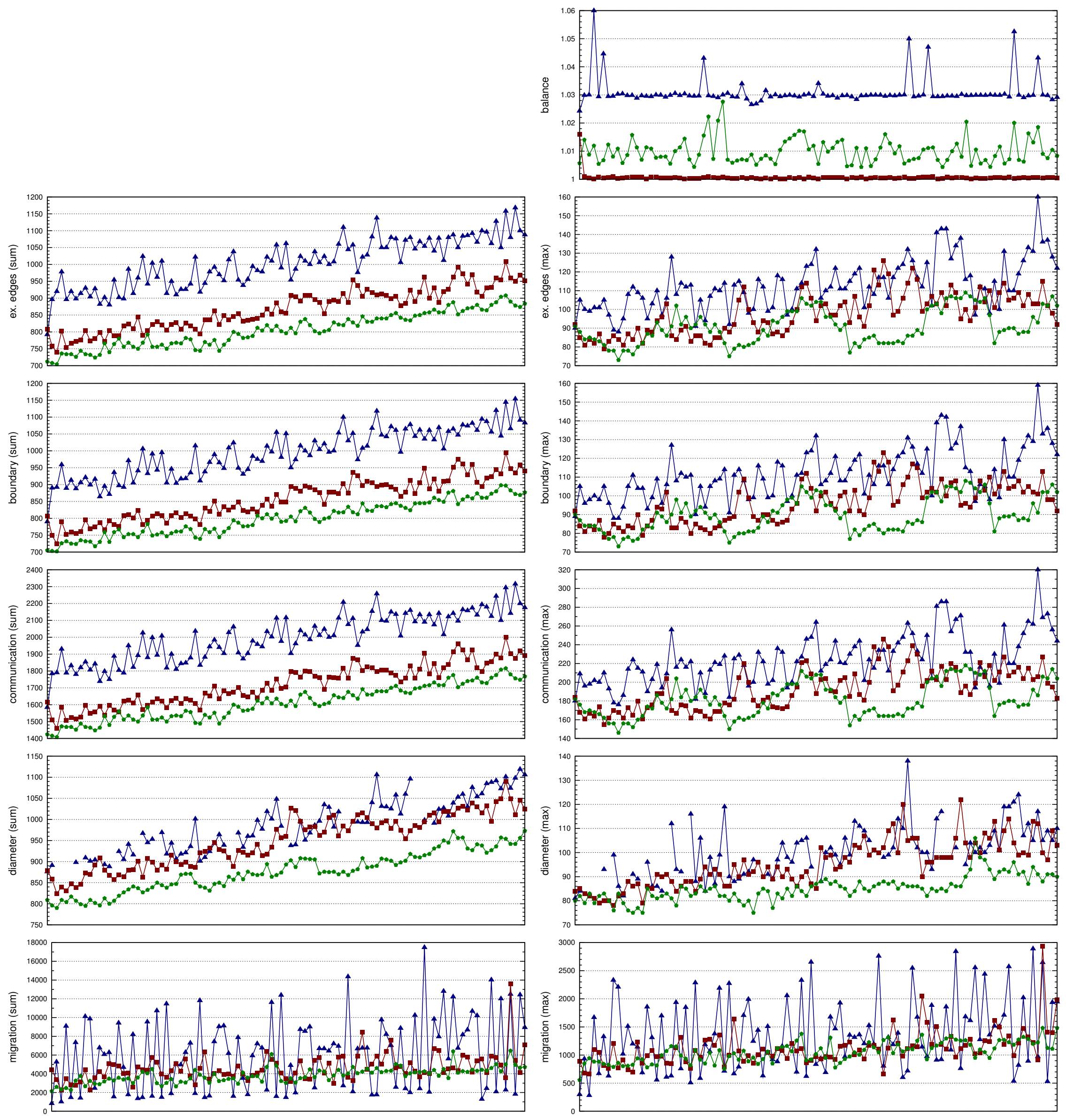


Figure 5.12: Results of all 101 frames of the 'slowtric' benchmark for Metis (blue triangles), Jostle (red squares) and Bubble (FOS/A) (green pentagons).

5.3.3 The First Order Scheme with Constant Draining

We are now introducing the FOS/C diffusion scheme. Like FOS/A, this scheme is also based on FOS and disturbed in every iteration. However, in contrast to FOS/A, the disturbance is not restricted to the non-empty vertices but performed on all vertices. Hence, negative loads become possible.

The FOS/C scheme executes two operations in each iteration. While the first one is the original diffusion, the second step introduces the disturbance by shifting a small load amount $\delta > 0$ from all vertices of the graph to some selected source vertices $S \subset V$. This disturbance can be described by the *drain vector* $d \in \mathbb{R}^n$, which is defined as

$$d_v = \begin{cases} \delta \cdot |V|/|S| - \delta & : \ v \in S \\ -\delta & : \ \text{otherwise} \end{cases}$$

This vector is added to the load vector resulting from the original diffusion step. Note that, since $\langle d, \mathbf{1} \rangle = 0$, this does not change the total amount of system load. The new scheme is formally described in the following definition.

Definition 3 (FOS/C). *Given a connected graph $G = (V, E)$ and a suitable constant α. Let $\delta > 0$ be the drain constant and d the corresponding drain vector. Let $S \subset V$ be the set of source vertices. In iteration k, $w_v^{(k)}$ denotes the load on vertex v and $x_e^{(k)}$ the exchange over edge e. Let $w^{(0)}$ represent the initial load situation. In each iteration k, the FOS/C scheme performs the computations:*

$$\begin{aligned} x_{e=(u,v)}^{(k)} &= \alpha \cdot \left(w_u^{(k)} - w_v^{(k)} \right) \\ w_v^{(k+1)} &= w_v^{(k)} - \sum_{e=(v,*)} x_e^{(k)} + d_v. \end{aligned}$$

In matrix notation, FOS/C can be written as

$$w^{(k+1)} = \mathbf{M} w^{(k)} + d$$

where $\mathbf{M}$ denotes $\mathbf{I} - \alpha\mathbf{L}$ like for FOS with $\mathbf{L}$ being the Laplacian matrix. $\mathbf{L}$ can be expressed as $\mathbf{L} = \mathbf{A}\mathbf{A}^T$, where $\mathbf{A}$ is the vertex edge incidence matrix of G. Note, that $\mathbf{L}$ does not have full rank.

In general, the existence of a solution of a linear system involving a Laplacian matrix $\mathbf{L}$ depends on the right hand side of the linear equation. The equation $\mathbf{L}x = b$ has a solution (and then infinitely many), iff $b \perp \mathbf{1}$. The next lemma states that the $||\cdot||_2$-minimal balancing flow can be computed by solving a system of linear equations.

Lemma 1 ([HB99]). *Consider the quadratic flow minimization problem*

$$\text{min! } ||f||_2 \textit{ with respect to } \mathbf{A}f = b$$

Provided that $b \perp \mathbf{1}$, the solution to this problem is given by

$$f = \mathbf{A}^T x, \textit{ where } \mathbf{L}x = b$$

To prove the convergence of FOS/C, we require the following observation.

Lemma 2. *Let $\mathbf{M}$ be a diffusion matrix and b be a vector perpendicular to $\mathbf{1}$. Then,*

$$\lim_{k\to\infty} (\mathbf{I} + \mathbf{M} + \mathbf{M}^2 + \cdots + \mathbf{M}^k)b = (\mathbf{I} - \mathbf{M})^{-1}b$$

Proof. Recall that $\mathbf{1}$ is an eigenvector to the simple eigenvalue 1 of $\mathbf{M}$. Since $b \perp \mathbf{1}$, i.e.$\sum_{j=1}^n b_j = 0$, it follows that $\lim_{k\to\infty} \mathbf{M}^{k+1}b = 0$. Hence,

$$\begin{aligned}
& \lim_{k\to\infty} (\mathbf{I} - \mathbf{M})(\mathbf{I} + \mathbf{M} + \mathbf{M}^2 + \cdots + \mathbf{M}^k)b \\
= & \lim_{k\to\infty} (\mathbf{I} - \mathbf{M}^{k+1})b = \lim_{k\to\infty} b - \mathbf{M}^{k+1}b \\
= & \; b
\end{aligned}$$

Therefore, $(\mathbf{I} + \mathbf{M} + \mathbf{M}^2 + \cdots + \mathbf{M}^k)$ is the inverse to $(\mathbf{I} - \mathbf{M})$ for $k \to \infty$ and any vector perpendicular to $\mathbf{1}$, so that the claim follows. □

Now, we are able to state the following theorem.

Theorem 1 (Convergence of FOS/C). *The FOS/C scheme converges for any arbitrary initial load vector $w^{(0)}$.*

Proof. Repeatedly applying the diffusion matrix to the initial load vector $w^{(0)}$, we obtain

$$\begin{aligned}
w^{(1)} &= \mathbf{M}w^{(0)} + d \\
w^{(2)} &= \mathbf{M}w^{(1)} + d = \mathbf{M}(\mathbf{M}w^{(0)} + d) + d \\
&= \mathbf{M}^2 w^{(0)} + (\mathbf{M} + \mathbf{I})d \\
&\vdots \\
w^{(k)} &= \mathbf{M}^k w^{(0)} + (\mathbf{M}^{k-1} + \cdots + \mathbf{M} + \mathbf{I})d
\end{aligned}$$

Due to lemma 2, this yields

$$\begin{aligned} w^{(\infty)} &= \mathbf{M}^{\infty} w^{(0)} + (\mathbf{I} - \mathbf{M})^{-1} d \\ &= \mathbf{M}^{\infty} w^{(0)} + (\alpha \mathbf{L})^{-1} d \end{aligned}$$

□

$w^{(\infty)}$ is composed of two parts. $\mathbf{M}^{\infty} w^{(0)}$ is the evenly balanced load $\overline{w}$ that FOS computes. Hence, the load differences only depend on $\mathbf{L}$ and d. This means that the solution of the disturbed scheme FOS/C can also be determined by solving a system of linear equations. Looking at the load in the converged state, the following observation can be made.

Corollary 1. *When converged, the load $w^{(\infty)}$ of FOS/C can be characterized as:*

$$\begin{aligned} w^{(\infty)} &= \mathbf{M} w^{(\infty)} + d \\ \Leftrightarrow (\mathbf{I} - \mathbf{M}) w^{(\infty)} &= d \\ \Leftrightarrow \alpha \mathbf{L} w^{(\infty)} &= d \\ \Leftrightarrow \mathbf{L} w^{(\infty)} &= d/\alpha \end{aligned}$$

Hence, the load distribution of FOS/C in its converged state can be determined by solving the system of linear equations $\mathbf{L} w^{(\infty)} = d/\alpha$.

For our purpose, the scaling factor α can be omitted. Due to the rank of the Laplacian matrix, the solution $\mathbf{L} w = d$ is only determined up to an additive constant reflecting the total load in the system. Hence, by solving the linear system of equations this constant cannot be determined, meaning that only the load differences between the vertices are unique.

In fact, in the converged state of FOS/C, all load that is moved onto the source vertices via the disturbance described by d has to be sent back in one iteration step. Hence, according to lemma 1, following corollary can be made.

Corollary 2. *The load differences*

$$f = \mathbf{A}^T w^{(\infty)}$$

in the converged state of the disturbed diffusion scheme FOS/C equal the

$$||\cdot||_2\text{-minimal flow } f$$

that balances the load vector d, sending from the vertices in S the load δ to every vertex in the graph.

This means that we can obtain the load differences of the solution of FOS/C with well known methods which calculate the $||\cdot||_2$-minimal flow that balances the initial load distribution d. These are for example the genuine diffusion schemes FOS and SOS, or solvers for linear systems of equations like the Conjugate Gradient (CG) or Algebraic Multigrid approach.

Once the balancing flow has been determined, the vertex loads $w^{(\infty)}$ of FOS/C can be assigned accordingly with a suitable chosen constant, e.g. such that $\sum_{v\in V} w_v^{(\infty)} = 0$. This choice also ensures that the load distributions computed for each partition have a common reference point and are therefore comparable. An alternative way, which generalizes this method and additionally improves the numerical stability, is discussed in the next subsection.

5.3.4 Numerical Stability by Influence Range Reduction

The load differences of the distribution in the converged state of the FOS/C scheme equal the $||\cdot||_2$-minimal flow that sends δ load from the source vertices to every other vertex in the graph. Hence, we can compute $w^{(\infty)}$ by solving the system of linear equations

$$\mathbf{L}w^{(\infty)} = d.$$

However, although this linear system has infinitely many solutions because of d being perpendicular to $\mathbf{1}$, the limited numerical precision causes errors and the solvers often diverge in practice. In the following, we present a modification of the linear system such that the involved matrix becomes positive definite instead of semi-definite which eliminates numerical problems.

To understand the meaning of the altered linear system, we first generalise the flow problem that we solve. We construct a new graph G_ϕ that is composed of the graph G and an extra vertex x that is connected with every other vertex of G. All edges $e \in E$ of G are assigned a capacity of $c_e = 1$ while the capacity of the edges incident to x are set to some constant $\phi > 0$. Again, we equally place load on the set of source vertices S, but now only let load drain from the extra vertex x. Since we minimize f according to the $||\cdot||_2$-norm, the load will not be sent directly to x, but also makes some 'detours' via other vertices in G.

The weight constant ϕ determines the spreading of the flow. If ϕ is large, it is 'cheaper' to send most load directly to x, while if ϕ is small, the costs of the 'detour' into the graph are compensated by less utilized edges incident to x that can be

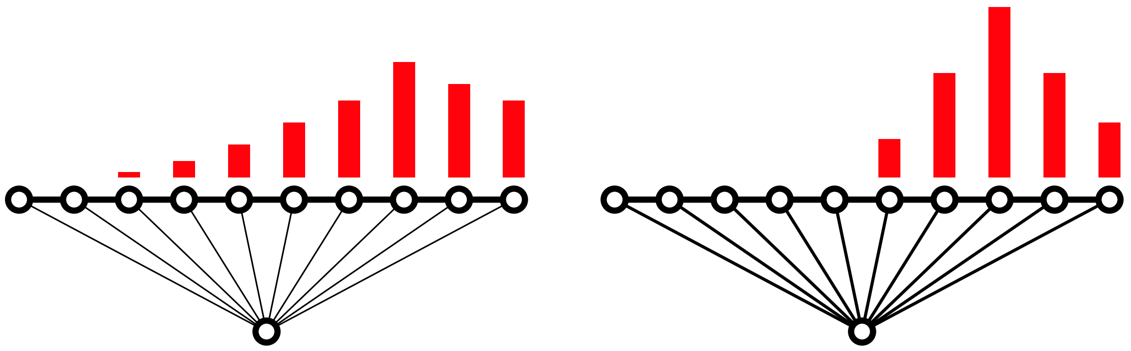


Figure 5.13: Flow towards x in case of a small (left) and large (right) parameter ϕ.

chosen. In the extreme cases, if $\phi \to \infty$, all load is sent directly to x, while if $\phi \to 0$, the $||\cdot||_2$-minimal flow will converge towards the solution obtained previously for the unmodified graph.

The modification weakens the influence of vertices that are far away from the set of source vertices and therefore not of interest to a partition. Of course, all edges still carry some load, since we compute the $||\cdot||_2$-minimal flow. However, due to the limited computational accuracy, the ones distant to the sources remain empty in practice as sketched in figure 5.13, displaying the flow over the extra edges depending on the choice of ϕ.

Formally, let $G = (V, E)$ be an undirected, connected graph. If we extend G by an additional vertex x and connect it to every other vertex with an edge of weight ϕ, we obtain the graph $G_\phi = (V \cup \{x\}, E \cup \{\{v, x\} : v \in V\})$ with edge weights $c_e = 1\ \forall e \in E$ and $c_{\{v,x\}} = \phi\ \forall v \in V$. The weighted Laplacian matrix $\mathbf{L}_\phi \in \mathbb{R}^{|V|+1 \times |V|+1}$ of G_ϕ is defined as $\mathbf{L}_\phi = \mathbf{A}_\phi \mathbf{C} {\mathbf{A}_\phi}^{\mathrm{T}}$, where $\mathbf{A}_\phi$ denotes the unweighted vertex-edge incidence matrix of G_ϕ, and the entries of the diagonal matrix $\mathbf{C} \in \mathbb{R}^{|E|+|V| \times |E|+|V|}$ are set to $(c_{ee}) = c_e$. If we denote the unweighted Laplacian matrix of G with $\mathbf{L} \in \mathbb{Z}^{|V| \times |V|}$, the weighted Laplacian matrix $\mathbf{L}_\phi$ of G_ϕ can be written as:

$$\mathbf{L}_\phi = \begin{pmatrix} \begin{pmatrix} & & \\ & \mathbf{L} + \phi\mathbf{I} & \\ & & \end{pmatrix} & \begin{matrix} -\phi \\ \vdots \\ -\phi \end{matrix} \\ \begin{matrix} -\phi & \cdots & -\phi \end{matrix} & |V| \cdot \phi \end{pmatrix}$$

The drain vector is extended accordingly. To subtract $|V| \cdot \delta$ load from the extra vertex x and add it equally to the set S of source vertices, d_ϕ looks like

$$d_{\phi_v} = \begin{cases} \delta \cdot |V|/|S| & : \ v \in S \\ -\delta \cdot |V| & : \ v \text{ is the extra vertex } x \\ 0 & : \ \text{otherwise} \end{cases}$$

Now, we compute the load distribution by solving the according linear system of equations

$$\mathbf{L}_\phi w_\phi^{(\infty)} = d_\phi.$$

The introduction of the extra vertex seems to introduce global information, since x is connected to every other vertex. Accordingly, the last row and column of $\mathbf{L}_\phi$ are completely occupied. Hence, some solvers like the algebraic multigrid cannot be applied directly. Furthermore, the rank of the modified matrix is still not full since $\mathbf{L}_\phi$ is still a Laplacian matrix.

However, solving an under-determined system of linear equations can be simplified [Kaa88]. Fixing as many entries of the solution vector as the dimension of the null space and deleting the corresponding rows and columns from the matrix and right hand side, one can obtain a solution by solving a fully determined system.

Hence, the numerical problems can be overcome by fixing the value of the extra vertex to be zero and deleting the row and column appended to $\mathbf{L}$ before. What remains is the addition of ϕ to the diagonal values of $\mathbf{L}$. This results in a symmetric positive-definite matrix $\mathbf{L}_\phi{}' = \mathbf{L} + \phi\mathbf{I}$, whose condition is controlled by the parameter ϕ. To obtain the load differences on the modified graph, we can now solve the fully determined linear system

$$\mathbf{L}_\phi{}' w_\phi{}' = d_\phi{}',$$

where $d_\phi{}'$ and $w_\phi{}'$ equal d_ϕ and w_ϕ without the entry for the extra vertex, respectively. Note that this preconditioning is well-defined by the notion of the extra vertex and interpretable as explained before.

After eliminating the entries for the extra vertex from the matrix, the structure of $\mathbf{L}_\phi{}'$ is the same as of $\mathbf{L}$, hence, the introduced global information is eliminated again. Furthermore, the entry for the extra vertex in the solution vector w'_{ϕ_x} is defined to be zero. Hence, it automatically acts as a common reference point and the load distributions for different partitions are now comparable without post-processing.

5.4 Balancing

Our experiments on a torus show that the proposed shape optimizing approach leads to almost equal partition sizes. The partitionings displayed in figure 5.14 illustrate the learning process. However, when being applied on inhomogeneous topologies, additional balancing becomes necessary. In the following, we describe two methods to improve the balance.

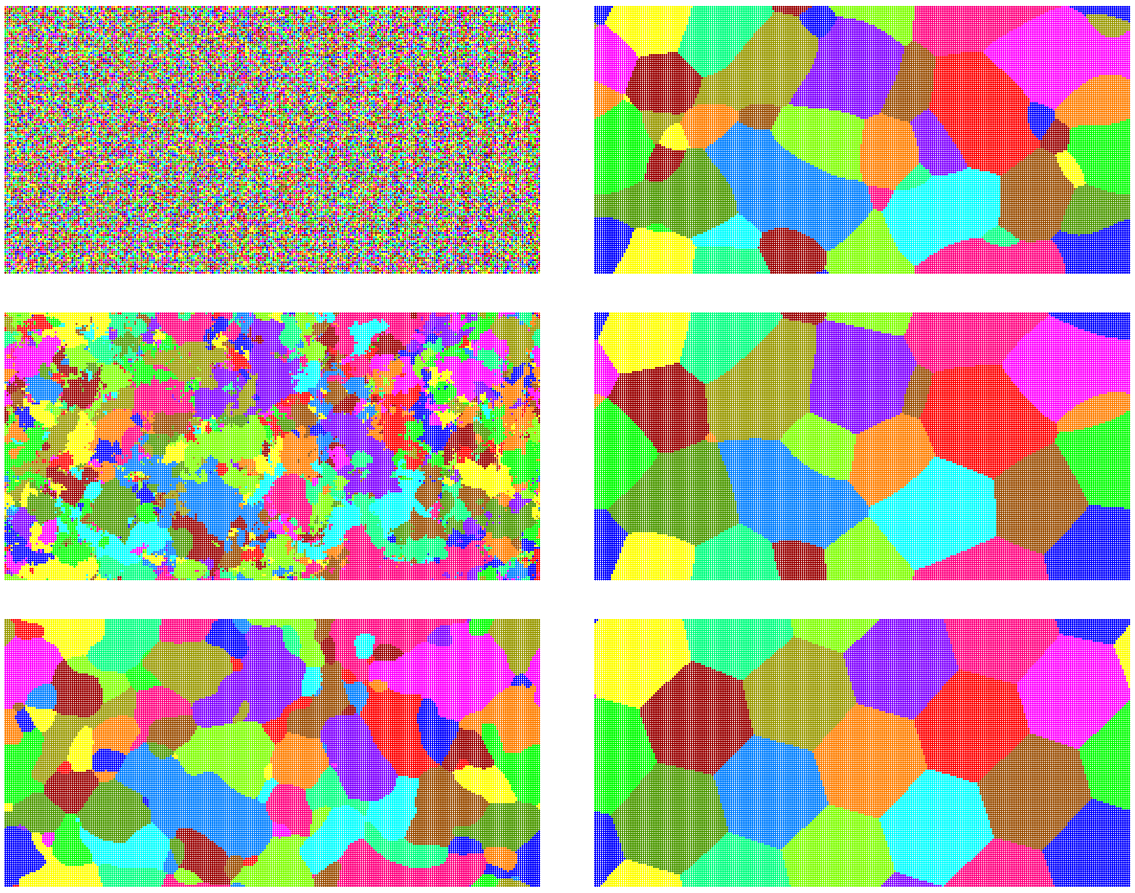


Figure 5.14: An initial random assignment on a 256×128 torus (top left) and the partitionings after 1, 2, 5, and 10 Grow/Consolidation operations, respectively. The final solution (bottom right), where no more changes occur, is obtained after 120 operations. In this example, a Move/Contraction is performed after every 10 invocations of Grow/Consolidation. Note that the final partitioning is composed of almost perfect hexagons, which is the regular shape with the most corners that can fully cover the 2-dimensional plane and therefore possesses the smallest boundary compared to its area.

5.4.1 Scale Balancing

A first idea to establish equal partition sizes is to adjust the amount of load that is placed on the partitions. If a partition is too small or too large, its total load amount is increased or decreased for the next learning step of the Bubble Framework. However, this requires the execution of additional diffusion processes and is therefore quite costly.

Instead of recomputing the load distribution, it is possible to adjust already existing solutions. Remember that the Bubble Framework assigns a vertex v to that partition with the highest load value

$$\pi_v = p : w_{\phi_v}^{p\prime} \geq w_{\phi_v}^{q\prime} \forall q.$$

As described in the previous section, we obtain the load vectors $w_\phi^{p\prime}$ by solving the linear system of equations $\mathbf{L}_\phi{}' w_\phi^{p\prime} = d_\phi^{p\prime}$ for each partition p, where the extra vertex x implicitly serves as common point of reference with a predefined load value of 0. Hence, since the computed load values correspond to a $||\cdot||_2$-minimal flow, changing the total load amount in the system is equivalent to scaling the final load vector $w_\phi^{p\prime}$.

The scaling factor ξ_p for a partition p is determined in an iterative process. Initially, ξ_p^0 is set to 1. If we denote the actual partition weights by ω_p and the total weight by Ω, we first calculate the desired balanced weight $\bar{\omega} = \Omega/P$. Next, we determine the scale coefficients $\zeta_p = (\bar{\omega}/\omega_p)^2$, which are limited to the range $[0.5, \ldots, 2]$. Now, the new scale factors ξ_p are computed as $\xi_p^{i+1} = \xi_p^i \cdot \theta + (1 - \theta) \cdot \zeta_p$. The parameter θ acts as damping factor and prevents that too small partitions become too large in the next step and vice versa. It is usually set to 0.9, and we perform twice the partition graph's diameter many iterations. The vertex assignment is then altered according to

$$\pi_v = p : \xi_p \cdot w_{\phi_v}^{p\prime} \geq \xi_q \cdot w_{\phi_v}^{q\prime} \forall q.$$

In many cases, the proposed simple scale balancing works very well. An example showing the maximal scaled load on the vertices and the resulting partitioning is given in figure 5.15. One can see that the balancing preserves the partition shapes. However, if the center vertices are not well distributed over the graph and a high imbalance occurs, it fails and cannot ensure equal sized partitions. In this case, we apply an additional greedy balancing step.

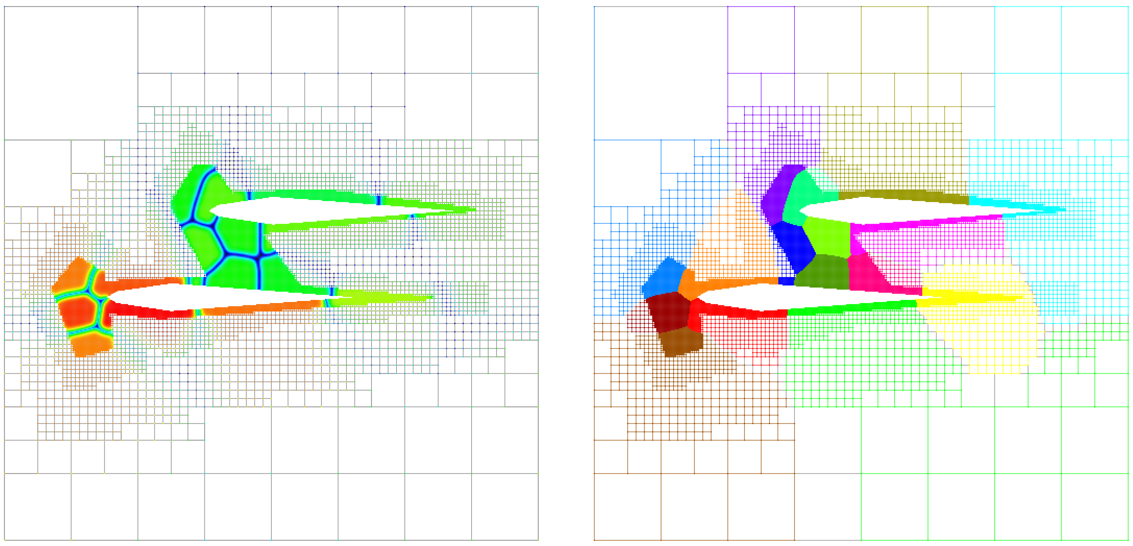


Figure 5.15: The maximal scaled load per vertex and the resulting partitioning.

5.4.2 Greedy Balancing

The initial random choice of center vertices can lead to situations where the partitions are not well distributed in the graph. Depending also on other aspects like the diameter of the partition graph, the scaling approach proposed in the last subsection is not able to properly balance the domain sizes. In extreme cases as shown in figure 5.16, the too large and too small domains are situated in opposite regions of the graph and a huge number of scaling iterations was required, since only neighboring partitions are affected first in the wave-like process.

If the scale balancing is not able to reduce the maximal partition overweight to less than 5%, we apply a greedy balancing. The greedy balancing is based on a $|| \cdot ||_2$-minimal balancing flow on the partition-graph that we compute as described in chapter 4. According to this flow, we now choose vertices for migration. A vertex v in partition p is considered, if the flow from partition p to partition q is positive, v is adjacent to a vertex u in q, and if the partition size of p is not too small. Among all considered vertices we move that one which causes the smallest error $w^{q\prime}_{\phi_v} - w^{p\prime}_{\phi_v}$ with regard to the load situation on the vertices. Once the vertex is moved, we upgrade the vertex values and reduce the balancing flow by 1. Note that these changes are local and only neighboring vertices are affected. The process is repeated until the flow on all edges of the partition graph is less than 0.5.

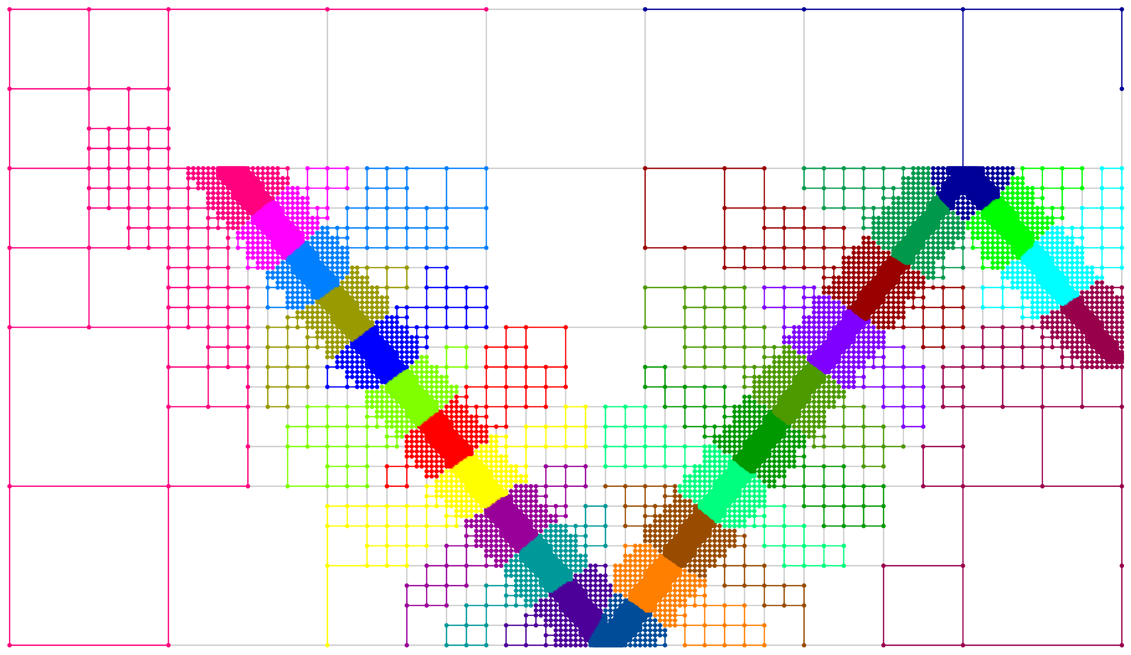


Figure 5.16: The graph 'stufe.10' partitioned into 24 domains as an example instance where balancing by scaling does not work well due to the graph structure. However, the greedy balancing can successfully be applied and align the partition sizes without deteriorating the solution quality.

5.4.3 Smoothing

To further enhance the partition quality, we perform a final post-processing step called smoothing. Once the learning process in the Bubble Framework has been completed, the smoothing further straightens the partition boundaries by migrating vertices that possess more neighbors in a different partition than neighbors in their own part. This procedure also eliminates possible artifacts in form of single isolated vertices that sometimes occur due to the limited numerical precision in the load calculation. Vertices with this property are considered, and the same migration process as described for the greedy balancing is performed.

In its current implementation, the smoothing does not consider the balancing. Hence, it might destroy an equal vertex distribution. However, from our experiments we conclude that because the fraction of affected vertices at the partition boundaries is relatively small, the imbalance usually does not increase by more than 1%.

5.5 The Flux Heuristic

Integrating the load distribution approach presented in the last section into the Bubble Framework, we obtain the Flux algorithm sketched in figure 5.17. Initially, it proceeds as described in figure 5.4 (lines 01 – 06), either choosing some center vertices randomly, or performing a consolidation step on either a random or the currently existing partitioning. The computation of the Move/Contraction (lines 09 – 11) and the Grow/Assignment and Grow/Consolidation operations (lines 14-16) is adapted. Instead of performing the diffusion iteratively, we compute the solution of FOS/C by solving systems of linear equations. For each partition p, we initialise the drain vectors $d_\phi^{p\prime}$ according to the current partitioning π. As described, load is placed on the set of source vertices containing either a single center vertex (line 09) or the whole partition (line 14). The linear systems are solved, and the new center vertices (line 11) or partitionings (line 14) are determined.

The balancing is inserted after the Grow/Assignment and Grow/Consolidation operations (line 17). It consists of the scale balancing and additionally performs a greedy balancing if the partition sizes vary by more than a given factor. Furthermore, the final solution is smoothed (line 18), before it is returned.

An interesting point is the lack of an explicit objective function. Except for the balancing process, the Flux heuristic does not contain any explicit directives concerning the metric to optimize. Hence, the diffusion process and the resulting load distribution are fully responsible for the solution quality.

The run-time of Flux greatly depends on the linear equation solver, and, of course, the parameters i and l. All other computations require only linear run-time or are negligible. Hence, an efficient linear solver is required, which is capable of processing several systems of equations with the same matrix $\mathbf{L}_\phi{}'$ but different right sides $d_\phi^{p\prime}$.

5.6 Implementation

To solve the involved systems of linear equations, we decide to apply a Conjugate Gradient (CG) solver. It is known to be a fast and reliable method to solve sparse positive definite linear systems. To improve the convergence rate of this iterative solver, several preconditioners are available. In the following subsections, we describe additional enhancements that address our specific problem instances.

Note that no work is necessary to set up the sparse matrix $\mathbf{L}_\phi{}'$, because it contains

```
Algorithm FLUX(G, i, l, $\phi$)
    in each iteration i
        if not $\pi$ exists
            case random-center
                $c$ = determine-random-centers(G)
            case random-partition
                $\pi$ = random-partitioning(G)
        if $\pi$ exists                                              /* contraction */
            parallel for each partition p
                set $d^{p\prime}_{\phi_v} = \begin{cases} \delta \cdot |V|/|\{v : \pi_v = p\}| & : \pi_v = p \\ 0 & : \text{otherwise} \end{cases}$
                solve $\mathbf{L}_\phi{}' w^{p\prime}_\phi = d^{p\prime}_\phi$
                $\pi_v = \begin{cases} p : w^{p\prime}_{\phi_v} \geq w^{p\prime}_{\phi_u} \forall u \in V \\ -1 : \text{otherwise} \end{cases}$
        in each loop l                    /* assignment & optional consolidations */
            parallel for each partition p
                set $d^{p\prime}_{\phi_v} = \begin{cases} \delta \cdot |V|/|\{v : \pi_v = p\}| & : \pi_v = p \\ 0 & : \text{otherwise} \end{cases}$
                solve $\mathbf{L}_\phi{}' w^{p\prime}_\phi = d^{p\prime}_\phi$
            $\pi_v = p : w^{p\prime}_{\phi_v} \geq w^{q\prime}_{\phi_v} \forall q \in \{1, \dots, P\}$
            $\pi$ = balance($\pi$, $w_\phi$)                                   /* balancing */
    return smooth($\pi$)                                             /* smoothing */
```

Figure 5.17: Sketch of the Flux heuristic.

the vertex degree plus ϕ on the main diagonal and off-diagonal entries for every edge. Hence, it matches our graph representation and the same sparse indexed data structures can be used.

Apart from CG, the Algebraic Multigrid approach is another efficient and state-of-the-art method for solving a linear system of equations. In our case, it might even appear to be predestined, since its costly preprocessing only depends on the matrix while solving the system afterwards is quite fast. Since we solve many systems with the same matrix $\mathbf{L}_\phi{}'$ that differ only in the right side $d^{p\prime}_\phi$, the preprocessing step has to be performed only once. First experiments are presented in [MMS06]. If it was possible to transfer the following techniques from the CG solver to the Algebraic Multigrid solver, we predict a significant run-time reduction.

Our implementation of the Flux heuristic for shared memory machines is straightforward. We follow exactly the formulation given in figure 5.17. Optionally, the linear systems can be solved by multiple threads. Usually, this version of our library is applied to compute an initial mesh distribution for a parallel simulation.

To repartition the data during an adaptive parallel simulation, we have implemented a distributed version of the Flux library that is based on the message passing interface (MPI). In this case we assume that the domains are already placed on the processing nodes. In the following, we describe important enhancements that further speed up the computations.

5.6.1 Partial Graph Coarsening

As described before, a vertex is assigned according to the maximum load value. Hence, we notice that due to the hill-like manner of the solutions only a part of them is relevant to the vertex assignment. Although the load distribution corresponds to an $||\cdot||_2$-minimal flow and every vertex receives some load from every partition, it is clear that in some areas the load from some partitions is negligible for the vertex assignments. The introduction of the parameter ϕ bars the load from spreading too far into the graph even more.

Hence, it is not necessary to compute the exact solution for all vertices of the graph, but only in important areas surrounding the respective partition, and refer to an approximation elsewhere. Of course, the important parts depend on the partition placements and are different for each of the P linear systems.

The observation can be exploited to both speed up the computations and reduce the memory requirements in a parallel implementation. Prior to the first computation, each domain creates a local level hierarchy. Similar as in the Party library, this hierarchy is based on a 2-approximation of a maximum weighted matching that is restricted to edges connecting local vertices. These are then combined to form the vertices of the graph in the next level. Now, the idea is to solve the linear systems with different solution accuracies on the domains by utilizing a composition of different hierarchy levels.

To solve a linear system, we first project the drain vector onto the respective vertices of the lowest hierarchy level and compute the load values there. Figure 5.18 (left) illustrates a solution for one partition on the lowest levels. One can see that the highest solution values can be found in and close to the originating domain.

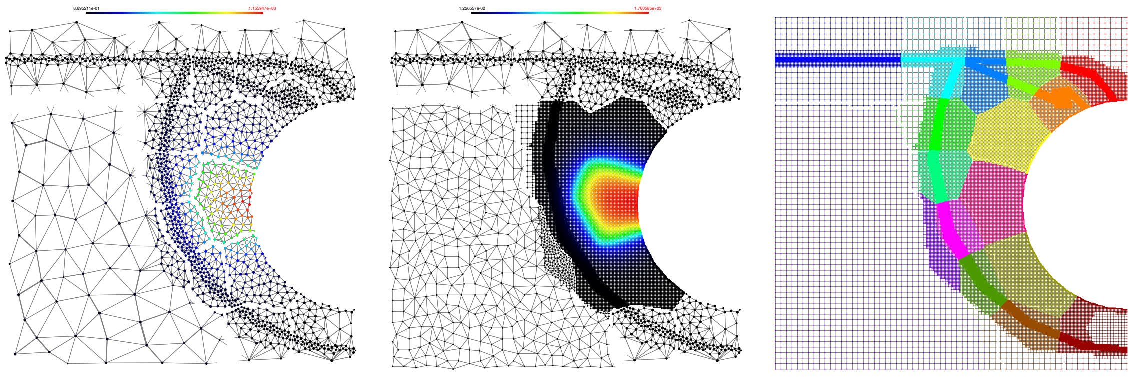


Figure 5.18: Vertex heights on the lowest levels (left) and the final solution with local accuracy on the respective levels (middle). The shown solution has been computed for the pink domain leading to the displayed partitioning (right). Edges between vertices of different domains (the initial partitioning) are cut.

Since the matching process preserves the graph structure, the solution on the lowest level is similar to the expected load distribution in the original graph. Hence, we are able to use it to determine the most relevant parts of the solution. Important domains within 10% of the highest occurring load value will be switched to a higher hierarchy level while the unimportant ones remain on the lowest one.

Figure 5.18 (middle) gives an example of a load distribution that has been calculated with varying accuracy. In the important regions of the graph, the linear system is solved on the highest hierarchy level, that is the original graph, while in areas further away from the respective domain lower levels are used.

Note that for each of the P linear systems a different part of the graph is important. Thus, the hierarchy levels that contribute to the respective solution differ for every system. To perform the communication, we require a data structure that allows to send and map the numerical data between the boundary vertices of different levels. Implementation details are given in the next paragraph. Furthermore, our library solves all systems simultaneously with a CG solver and we concatenate the data sent by all P instances. Hence, in every iteration of the solver only one message is sent in every direction between two neighboring partitions.

A Distributed Graph Data Structure A distributed linear system is usually realized via local copies of the numerical values stored at the partition boundaries, which are updated in every iteration of the solver. Figure 5.19 (left) sketches an example.

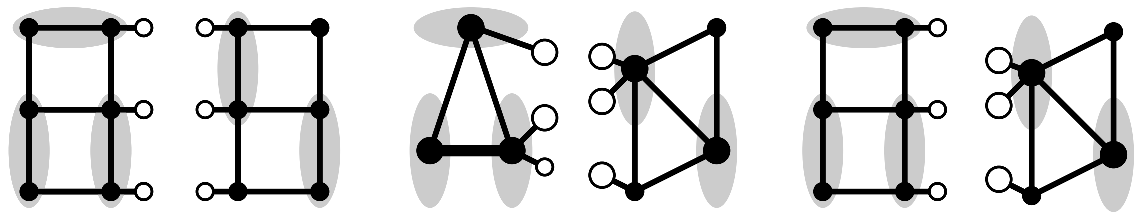


Figure 5.19: Sketch of the distributed graph data structure. A graph with 12 vertices is divided into two partitions. The halo vertices mirroring their originals at the boundaries are displayed white and the matching on the higher level is indicated in gray. Coarsening the original graph shown left results in a weighted graph displayed in the middle. The combination of two different levels is drawn right.

According to the domain decomposition, every processing node contains its part of the graph plus local copies of the adjacent vertices of neighboring partitions. These additional vertices are called the *halo*. Note, that we store a separate halo vertex for every cut edge. Hence, the same vertex can be mirrored several times.

Mirroring the same vertex to store one copy for each edge has several advantages if we further ensure that all vertices, as well as their numerical data, are stored in an array first sorted by the vertex domain, and additionally, that the halo vertices are in the same order as their represented originals in the neighboring domain. Every halo vertex is of degree one. Hence, to communicate the load values between the partitioning boundaries, we only have to run over the halo vertices and copy the numerical data of their only neighbor into the message buffer. The buffer is then communicated and because the halo vertices on the receiving domain are subsequently stored in memory, the message buffer can just be copied in a single block. This property still remains after combining vertices on the next hierarchy level as illustrated in figure 5.19 (middle). Hence, no global identifiers are required.

Using different levels in one linear system raises the question of how to obtain the halo values of vertices that do not exist in the current level of the neighboring partition as shown in figure 5.19 (right). Our implementation proceeds as follows. To transfer the solution values between different levels at the domain boundaries, the domain using the higher level is in charge of either combining the numbers to be sent or distributing them on receiving, according to the calculated matching. Figure 5.20 sketches an example in more detail. The halo values for the lower hierarchy level are obtained by a weighted summation of the numerical data from the higher level.

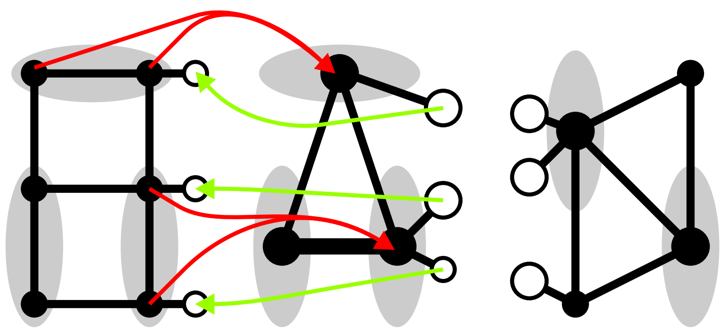


Figure 5.20: Sketch of the communication process between different hierarchy levels. The red arrows indicate the combining operation while the interpolation is drawn in green.

As an example, if two vertices v_1 and v_2 with weights s_1 and s_2 and solution vectors w_1 and w_2 are matched to $v_{1,2}$, the new vertex $l_{1,2}$ with weight $s_{1,2} = w_1 + w_2$ in the lower level obtains the value $w_{1,2} = (s_1 * w_1 + s_2 * w_2)/s_{1,2}$. For the reverse direction, the values for the higher hierarchy level are simply copied from their representatives in the lower level. Note that this described transformation is applicable not only between two consecutive levels, but between arbitrary ones in the hierarchy.

Although we now have to solve two linear systems per partition, a small one on the lowest hierarchy levels and the second one on the mixed levels, less run-time is required in total compared to solving one system on the original graph, if the number of partitions is large enough. The lowest levels of the hierarchy are very small and can be processed quickly. The additional time spent in this computation is compensated by the reduction of the system sizes in the second computation.

5.6.2 Domain Decomposition and Domain Sharing

The domain decomposition (DD) approach is the usual way to distribute a graph representing a system of linear equations on a parallel computer. Following this practice, the implemented preconditioned CG solver requires three communications per iteration, one matrix communication that updates the halo values and two scalar products. Hence, the number of messages is proportional to the number of iterations, which typically grows with the system size.

Since we solve P linear systems concurrently, an alternative distribution scheme for the computations exists. Instead of having every node process the chosen hierarchy level of its own domain for each of the P systems, it is possible to assemble one complete linear system on every processor. The systems are then solved locally without any communication, and finally the solution is sent back to the domains. We call this approach *domain sharing* (DS).

Domain sharing requires copies of all domains on every other node, which usually is impossible due to the involved memory requirements. However, we have seen that an accurate solution is not required in all areas of the graph, especially if the number of partitions is large. Hence, mainly lower levels of the hierarchy are requested what reduces the memory requirements significantly.

5.6.3 Multilevel Strategy

The multilevel strategy (refer to section 2.5) is a very powerful approach to determine good partitionings quickly. Hence, we can utilize the level hierarchy not only to reduce the size of the linear systems but also to speed up the learning process. Instead of running the learning process in the Bubble Framework on the original graph, we apply it on the intermediate levels and project the solution onto the next higher level to obtain a good initial partition placement. Experiments show that only one Move/Contraction operation per level is sufficient to obtain solutions that are comparable to other state-of-the-art partitioning libraries [MMS06]. However, when performing a repartitioning, the domains and the resulting center vertices are already well-distributed in the graph. Hence, in this case the multilevel strategy is obsolete and because we focus on this kind of application, learning on multiple levels is currently not activated in our parallel library.

5.7 Experiments

This section contains some of the experimental results we obtain with the Flux library. We first introduce the metrics and norms that we consider to express the quality of a partitioning. Next, we describe the benchmark sets. Besides the known static graph instances from the graph partitioning community, we also create new test instances containing graph sequences. These sequences reflect dynamic changes of a graph and allow to record repartitioning results. We continue with the presentation of some selected experiments which cover different settings of Flux and a comparison with Metis and Jostle. Finally, we report about the integration of Flux into the parallel adaptive finite element simulation tool PadFEM.

5.7.1 Metrics and Norms

To measure the quality of a partitioning, a number of metrics are possible. The traditional one is the edge-cut, that is the number of edges between different partitions, but it is known that this usually does not model the real costs [Hen98]. Depending on the application, some of the metrics might be more important than others. Hence, we consider a number of them which can be described as follows and are measured per partition p.

External edges Number of edges that are incident to exactly one vertex of partition p, meaning that they are in-between two partitions:

$$\mathrm{ext}(p) = |\{e = (u, v) \in E : \pi(u) = p \wedge \pi(v) \neq p\}|$$

Boundary vertices Number of vertices of partition p that are incident to at least one external edge:

$$\mathrm{bnd}(p) = |\{v \in V : \pi(v) = p \wedge \exists u \in V : \pi(u) \neq p \wedge (v, u) \in E\}|$$

Diameter The longest shortest path between two vertices of the partition p. Infinity, if the partition is not connected.

Outgoing migration Number of vertices that are migrated from partition p to another partition.

Incoming migration Number of vertices that are migrated from another partition to partition p.

The number of vertices at the partition boundary reflects the amount of information to be sent to a neighboring partition more precisely than the edge-cut. Usually, the boundary vertices are mirrored in the neighboring partition and updated before every iteration, such that the locally operating algorithms can access their data. Hence, data has to be communicated only once for each vertex, even if it has more than one pair in that partition. Note that the number of boundary vertices does not exactly match the amount of information to be sent due to vertices that are adjacent to more than one neighboring partition. However, only a few such vertices exist and in our experiments these numbers are negligible.

Additionally, the quality of a partitioning depends on its balance. A less balanced solution does not necessarily cause problems during the computation, but of course allows other metrics to improve further.

The metrics recorded for every partition are summarized according to three different norms. Given the values $x_1, \dots, x_P$, these norms are defined as follows:

Summation Norm

$$||X||_1 := x_1 + \ldots + x_P$$

Euclidean Norm

$$||X||_2 := (x_1^2 + \cdots + x_P^2)^{1/2}$$

Maximum Norm

$$||X||_\infty := \max_{i=1..P} x_i$$

The $||\cdot||_1$-norm (summation norm) is a global norm. The global edge cut belongs into this category (it equals half the external edges in this norm). In contrast to the $||\cdot||_1$-norm, the $||\cdot||_\infty$-norm (maximum norm) is a local norm only considering the worst value. This norm is favorable if synchronized processes are involved. The $||\cdot||_2$-norm (Euclidean norm) is in between the $||\cdot||_1$ and the $||\cdot||_\infty$-norm and reflects the global situation as well as local peaks.

5.7.2 Benchmarks

As mentioned in chapter 2, some collections of graphs exist, which serve as a benchmark set in a large number of publications. To compare Flux with the established graph partitioning libraries, we use a subset of these test instances. Our presentation includes the graphs that are listed in table 5.1.

While a graph partitioning benchmark only consists of single graphs, the evaluation of load balancing heuristics requires sequences of them. Unfortunately, there is no such benchmark collections available. Thus, we have developed a basic though very expandable generator which produces graph sequences with properties that are very similar to those of adaptive two dimensional FEM computations [MS05]. The creation of the test sets is explained in the following.

Table 5.1: Graphs in the test set and some of their properties.

			degree				
Graph	$\lvert V\rvert$	$\lvert E\rvert$	min	avg	max	diameter	origin
airfoil1	4253	12289	3	5.78	9	43	FEM 2D
biplane.9	21701	42038	2	3.87	4	102	FEM 2D
crack	10240	30380	3	5.93	9	84	FEM 2D
grid100	10000	19800	2	3.96	4	198	FEM 2D
ocean	143437	409593	1	5.71	6	229	FEM 3D dual
shock.9	36476	71290	2	3.909	4	237	FEM 2D
stufe.10	24010	46414	2	3.87	4	54	FEM 2D
wave	156317	1059331	3	13.55	44	38	FEM 3D

Graph Sequences and Transitions

A graph sequence reflects the changes of the discretization caused by the mesh refinement and coarsening procedure. Each graph, also called frame, is the dual to the static mesh at that point when the load balancing algorithm is started. Performing the mesh adaptation in parallel, new elements usually belong to the domain they have been created in. To be able to provide this information independently of a given partitioning, each of the inserted elements must be assigned to one element of the preceding graph. This allows to place the new objects onto the same processor as their predecessors and therefore to model the parallel behavior, even if the actual partitioning is not known. Since elements can also be removed and the order of existing elements can change, it is necessary to assign a predecessor to every element of the new mesh. We refer to this data as transition information.

Figure 5.21 gives an example. On the left, two consecutive frames of the mesh are shown. Their dual graph, which forms the input for the load balancers, is displayed next to them. Note, that the graphs' vertices are numbered continuously from 0 to $|V|-1$ and that a vertex's number can be different in the following frame, although it itself has not been altered.

With an artificially generated sequence, the evaluation of a new algorithm can be performed much faster than if it is run inside a real application since no mathematical computations or meshing operations have to be performed. Being independent of the availability of models and mathematical code is another important advantage. Furthermore, the large amount of numerical data in real simulations prevents the use of common hardware that now is sufficient for the tests.

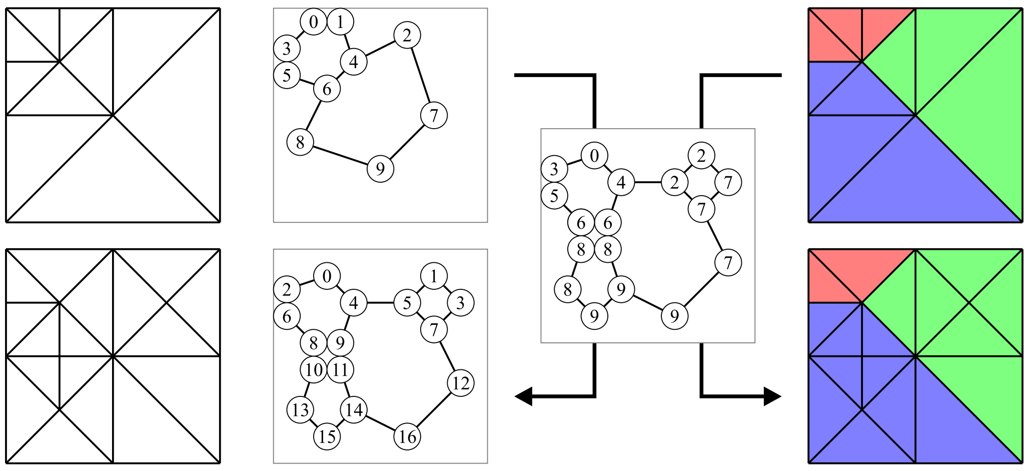


Figure 5.21: Two consecutive frames of a mesh sequence (left) and their corresponding dual graph used for partitioning (middle left). During the creation process, the predecessor of each node in the new graph has been recorded (middle right). With this additional transition information any given partitioning can be mapped (right).

The minimal required input for a dynamic benchmark consists of a number of graphs and the interleaving transition information. Optionally, it is possible to provide some more data to draw the graph or even the originating mesh. Visual evaluation of the results sometimes reveals important deficiencies and possible improvements and should not be underestimated. Vertex coordinates for the graph, which are a requirement when testing geometric approaches, can be provided easily.

A Basic Sequence Generator

During the development process of a load balancing heuristic, many single features and their combinations have to be tested. Some of them only consider special cases that rarely occur in real simulations but that are nevertheless important to be handled correctly to ensure the stability of the partitioning and load balancing library. To be able to generate the required constellations, we have created a basic tool that outputs artificial mesh sequences.

The applied mesh generation process is based on a known refinement scheme. Starting with a 2-dimensional square divided into two triangles, the latter can be refined by always inserting a new vertex at a triangle's base (which is the longest edge). Since this scheme is regular, we obtain meshes with angles of 45 and 90 degree only. Before a triangle is split, we check if this leads to a T-intersection.

In this case, the affected larger triangle is split first, which is a recursive process. Merging is also possible if none of the anticipating triangles is required anymore to avoid T-intersections. Note, that the whole refinement tree is stored at all time and that the triangles of the mesh are the leaves of this tree.

To determine which triangle to split, we insert so called attractors. These are currently implemented as points, but other geometric shapes are possible. All triangles compute the ratio of their area and the distance to the closest attractor. These values are stored in a priority queue, such that always the triangle with the largest ratio can be determined and be selected for refinement next. A second queue holds triangles for merging, which is also performed based on the distance/area ratio.

Furthermore, we use a geometry to cut areas out of the mesh. If a triangle is fully covered by the geometry, it is marked inactive and will no longer be part of the mesh. Currently, the geometry is limited to circle sets, but extensions are also possible here.

The means described so far allow us to generate a static mesh. To create a sequence of meshes with small changes between them, traces are assigned to both, attractors and the geometry. A trace describes how an object changes its position over time. Furthermore, the attractors can vary their weight and the global refinement threshold can be adjusted via time-depended functions. After the adjustments have taken place, all values are updated, and triangles that are now too large are split or merged if too small.

It is obvious that the generated meshes are unusable for mathematical computations. A closer look at the border of the circles in figures 5.22 to 5.31 reveals that the circle surface is not approximated smoothly at all what for example will cause a fluid dynamics simulation to fail. However, since the graph partitioning and load balancing algorithms only consider the dual graph of the mesh they are not affected. Our experiments with meshes generated this way show that the created instances reflect the properties and behavior of real problem instances well enough to lead to meaningful results.

To record the transition information described in the previous section, we proceed as follows. Before each time step, all triangles of the mesh are assigned a serial number. The dual graph is then saved on disk with the vertices stored in their assigned serial order. If a triangle is split, both children of this triangle inherit its number. When merging two triangles, the number of the left triangle is assigned. After the mesh adaptation, the old serial numbers now contain the information about the predecessor of each vertex in the previous graph. This forms the transition data. New serial numbers are assigned to the adapted graph, and the collected transition data is stored in a file according to the new vertex enumeration.

Due to geometry movements, it can happen that triangles marked inactive become active again, e.g., after a circle has changed its position. These leafs do not have a valid transition number. To handle this case, all triangles with valid numbers propagate their data in the tree towards the root with the left child of an inner node having priority. In a second step, triangles without valid numbers are assigned the same number as their parent. This ensures that the uncovered triangles will be placed in the same partition as one of their neighbors.

To display meshes graphically, we create additional files that contain the three triangle coordinates according to the vertices of each dual graph. With this additional geometric information we are able to generate the postscript images that are presented in the following.

Test-set Sequences

The included experiments are based on ten graph sequences. Frames 0, 25, 50, 75, and 100 of the sequences are displayed in figures 5.22 to 5.31, respectively. In the first four settings, the geometry (red) is constant and only the refinement of the mesh elements varies. The 'refine' benchmark refines the mesh around a circle positioned in the center of the simulation area. In the 'change' sequence, also some elements are combined and the mesh is coarsened around the second circle. In the 'heat' benchmark, the geometry covers large parts of the simulation area, leading to some long and thin simulation spaces. The refined area moves from the top left corner to the bottom right. The 'ring' and 'ring2' sequences rotate the refined area in a ring like geometry, and 'ring2' additionally contains an object therein. In the 'circles' benchmark, three circles of different size rotate in a row around the center. The smaller a circle, the finer is the mesh discretization. Figures 5.28 and 5.29 rotate the same circles, too, but they are placed equidistant. In the 'fasttric' benchmark, the movement speed is twice as high as in the 'slowtric' case. In the benchmarks 'bubbles' and 'trace' some circles cross the simulation area. While in the first case the area is refined according to the distance to the geometry, the 'trace' benchmark delays the coarsening such that many elements remain on the path of the geometry.

All instances shown are designed to contain about 20000 elements. Depending on the setting, this number varies over time due to the mesh adaptation. Additionally,

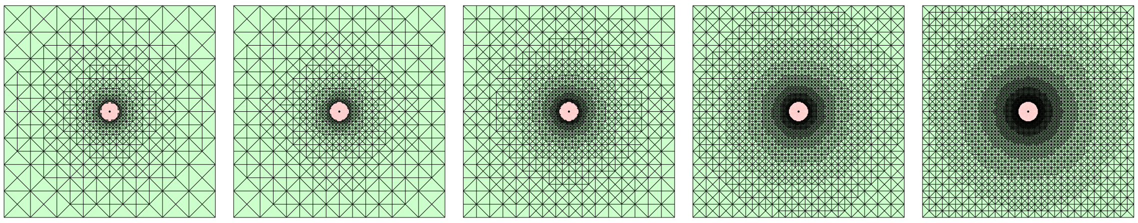


Figure 5.22: Frames 0, 25, 50, 75, and 100 of the ’refine’ sequence.

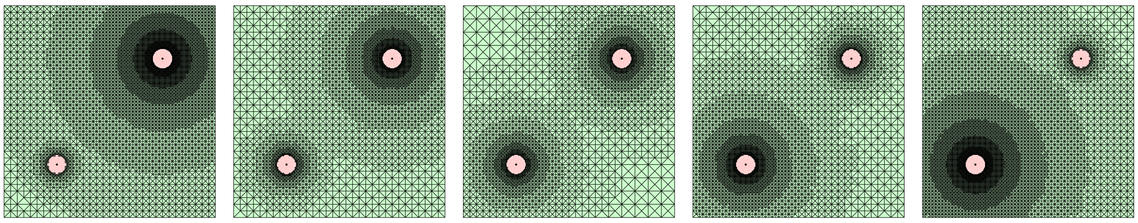


Figure 5.23: Frames 0, 25, 50, 75, and 100 of the ’change’ sequence.

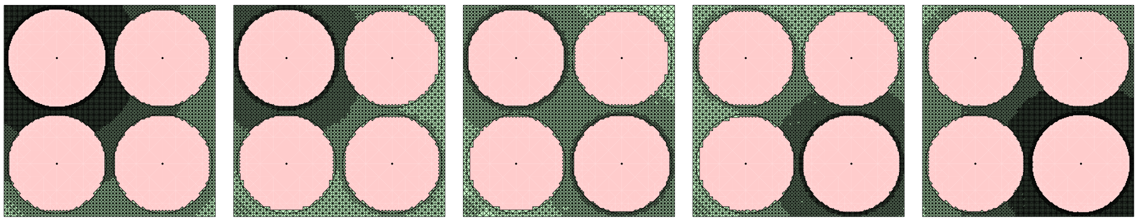


Figure 5.24: Frames 0, 25, 50, 75, and 100 of the ’heat’ sequence.

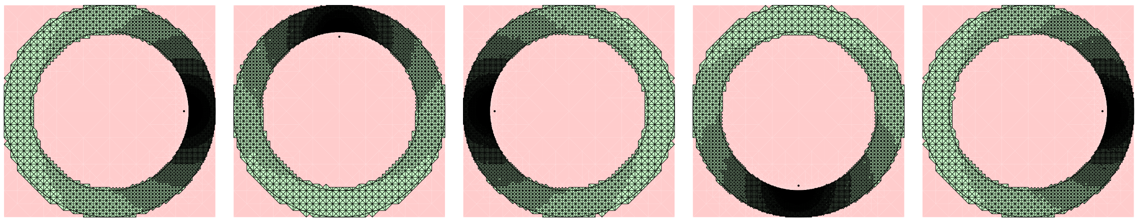


Figure 5.25: Frames 0, 25, 50, 75, and 100 of the ’ring’ sequence.

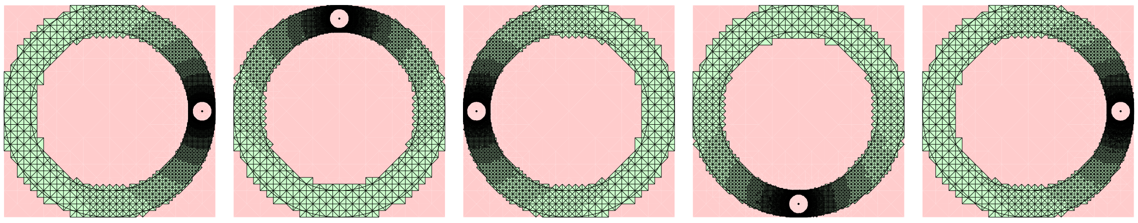


Figure 5.26: Frames 0, 25, 50, 75, and 100 of the ’ring2’ sequence.

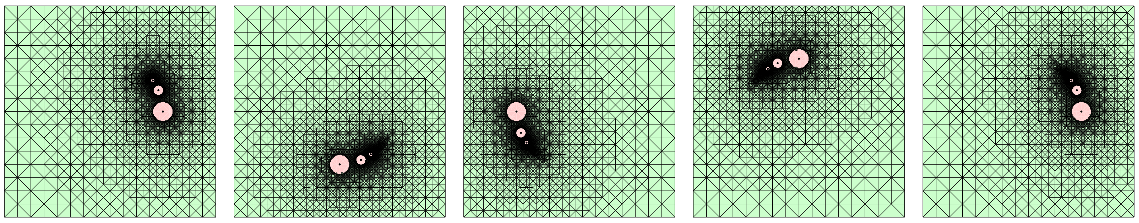


Figure 5.27: Frames 0, 25, 50, 75, and 100 of the 'circles' sequence.

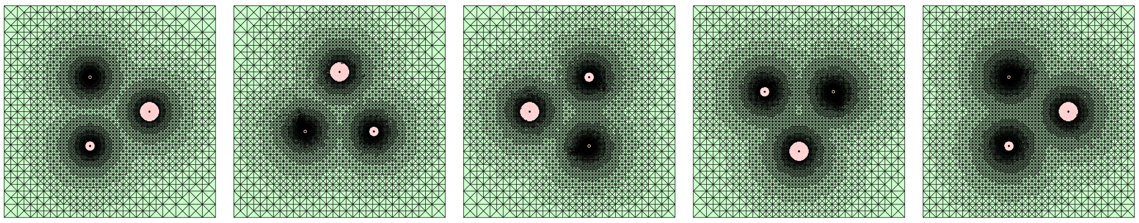


Figure 5.28: Frames 0, 25, 50, 75, and 100 of the 'slowtric' sequence.

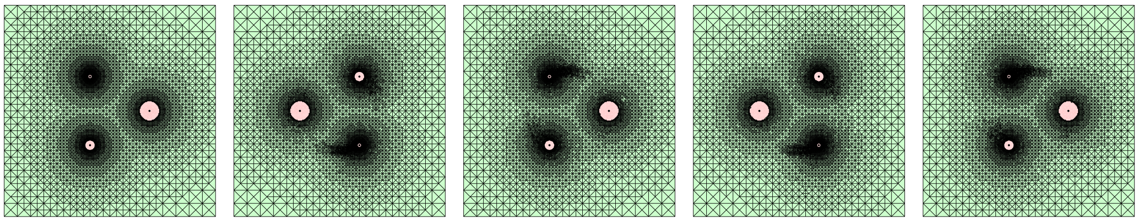


Figure 5.29: Frames 0, 25, 50, 75, and 100 of the 'fasttric' sequence.

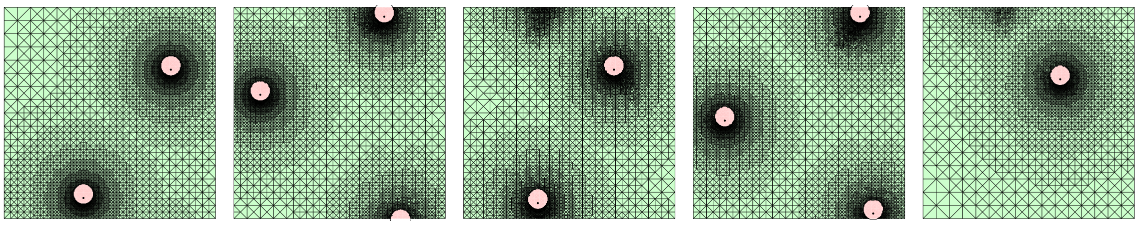


Figure 5.30: Frames 0, 25, 50, 75, and 100 of the 'bubbles' sequence.

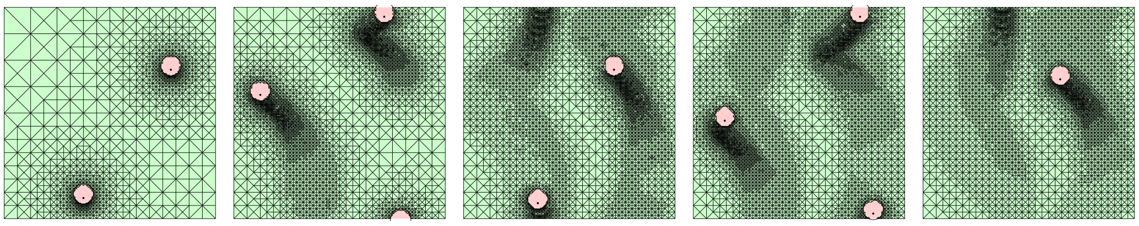


Figure 5.31: Frames 0, 25, 50, 75, and 100 of the 'trace' sequence.

we created larger versions of some benchmarks by tightening the area restrictions of the triangles to 1/10 of the usual value, which results in graph sequences of 100000 to one million vertices per frame.

5.7.3 Numerical Results

In this subsection we present numerical data from our experiments with the Flux library. We first look at the partitioning problem, measure the influence of different parameter settings, and compare our results with those computed with other libraries. We continue with similar experiments for the repartitioning problem and report on the results we obtained on the graph sequences. Finally, we examine the run-times of the distributed implementation in a parallel computing environment.

Graph Partitioning

The most important parameters of the Flux heuristic are the number of learning steps (iterations and loops) and the influence range reduction ϕ. To demonstrate their meaning, we run a number of experiments on the graphs of our test sets and vary only one of these values. Due to the similarity of the results, we limit the presentations to 12 or 16 partitions, and show only two detailed examples each. The experiments are performed with our sequential library on a 3.0 GHz Pentium 4 computer equipped with 2 GB of RAM. According to the evaluation scheme from section 3.1, each bar displays the average value of 10 independent runs with a large mark, the standard deviation of the values with a wide bar, the minimum and maximum values with thin bars and the result for the first run with a small mark.

Influence of ϕ From the theoretical analysis presented in section 5.3.4, we know that the parameter ϕ influences the run-time as well as the partition quality. A large ϕ reduces the number of iterations that are required in the CG solver due the modified matrix $\mathbf{L}'_\phi$ and bars the load from spreading far into the graph, which leads to a different solution. Furthermore, it is clear that ϕ has to be in a certain range. If too large, vertices in the graph will remain empty, and a value too close to zero often leads to divergence in the CG solver due to the limited numerical precision. Hence, we run experiments with $\phi \in \{0.05, 0.01, 0.005, 0.001, 0.0005\}$, respectively. The initial partition centers are determined randomly (Init/Random Center). The results for the 'biplane.9' and 'ocean' graph are presented in figure 5.32 and 5.33.

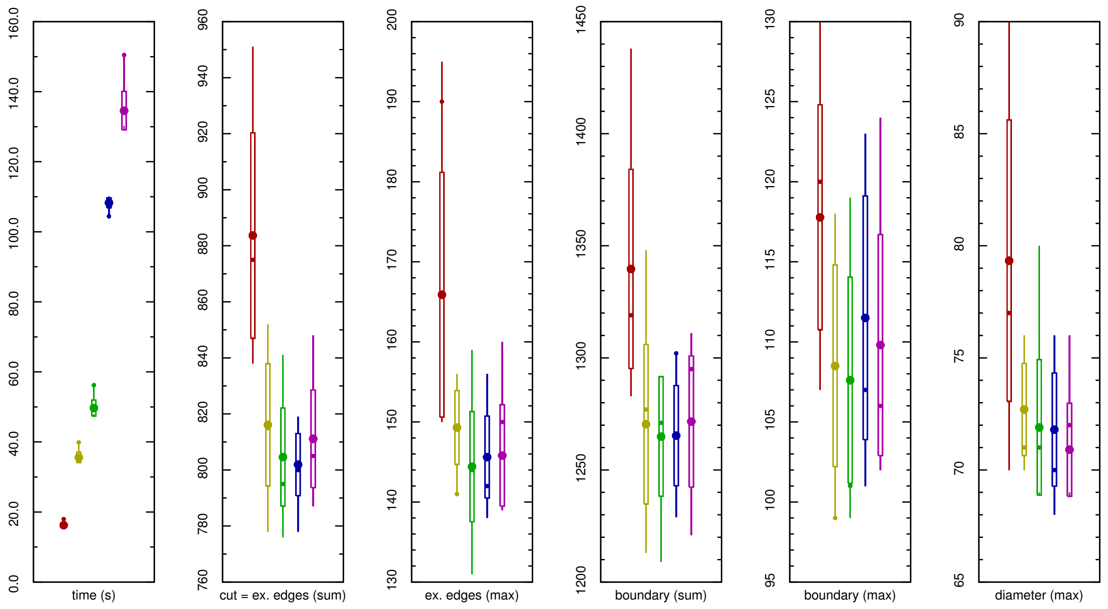


Figure 5.32: Partitioning the 'biplane.9' graph into 16 parts running 4 iterations and 4 loops for different values of ϕ: 0.05 (red), 0.01 (yellow), 0.005 (green), 0.001 (blue) 0.0005 (magenta).

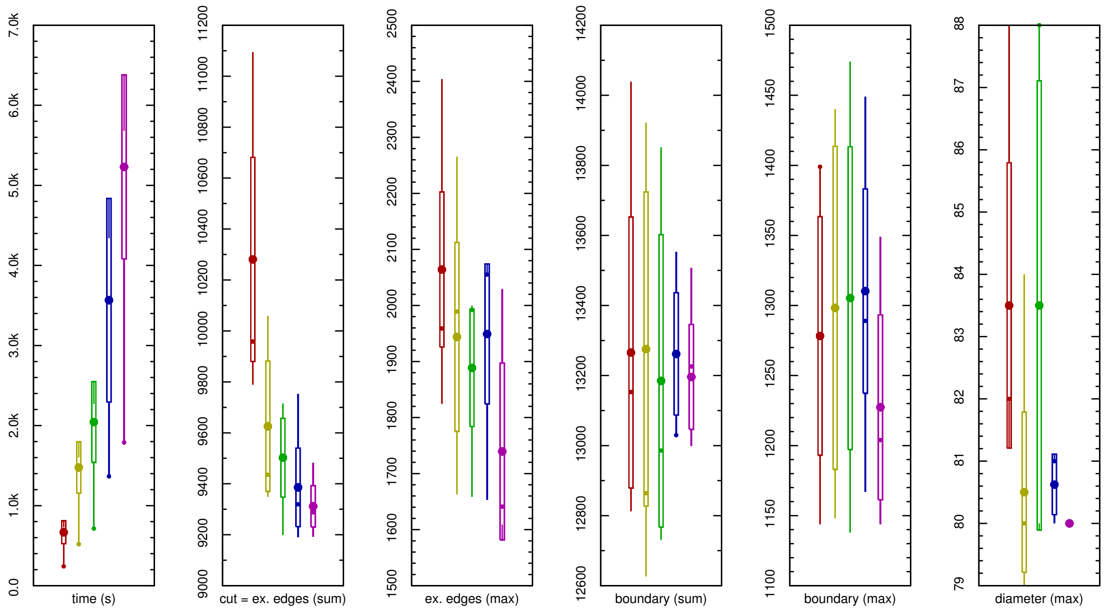


Figure 5.33: Partitioning the 'ocean' graph into 16 parts running 4 iterations and 4 loops for different values of ϕ: 0.05 (red), 0.01 (yellow), 0.005 (green), 0.001 (blue) 0.0005 (magenta).

As in each of the following charts, the run-time is displayed first. It meets the expectation and a smaller ϕ increases the number of CG iterations and therefore the execution time. An interesting point is that Flux performs fastest on the original graph representation (small dot) in case of the 'ocean' graph, while the opposite can be observed for the 'biplane.9' graph. We explain this behavior with caching effects, because 'ocean' is larger than 'biplane.9' and therefore its data does not fit into the processor cache. However, this might point to a possible optimization.

Next, the number of external edges is displayed in summation and maximum norm. In general, increasing ϕ improves the partition quality, although there are some exceptions to this rule as for example in case of the 'biplane.9' benchmark with a small ϕ, where no better solution can be found. Enlarging ϕ also reduces the variance of the results and the algorithm becomes more reliable. Another observation is that the maximum norm of the edge-cut shows some correlations with the number of boundary vertices shown in the following columns.

For the number of boundary vertices, the influence of ϕ is less obvious. For the two displayed examples, the total boundary is more or less equal for several values of ϕ and in case of the 'ocean' graph the maximum boundary does increase for any smaller ϕ except 0.0005. Note, that the variance of the values is larger than the differences between their means. Hence, a conclusion is difficult to make. However, also considering the results of the remaining graphs of our test set, we state that a smaller ϕ usually leads to less boundary vertices in both norms.

Concerning the diameter, the same rule as for the external edges and the boundary can be observed, although there is again an exception in case of the 'ocean' benchmark when choosing $\phi = 0.005$.

Summarizing these experiments, we observe that the partition quality usually improves in all metrics and norms with a lower choice of ϕ. However, as already mentioned, choosing ϕ too small results in numerical instabilities, hence we decide to take 0.001 as the default.

Number of Iterations and Loops Altering the number learning steps has a direct influence on both, the run-time and the solution quality. Performing more iterations or loops results in a longer execution time as shown in figures 5.34 and 5.35 for the 'biplane.9' and 'ocean' graph. Also, the already described caching effect is noticeable. From the values of the 2/2, 4/2 and 8/2 settings shown for the 'biplane.9' and 'ocean' graphs, we observe, that running more iterations usually decreases the edge-cut, the

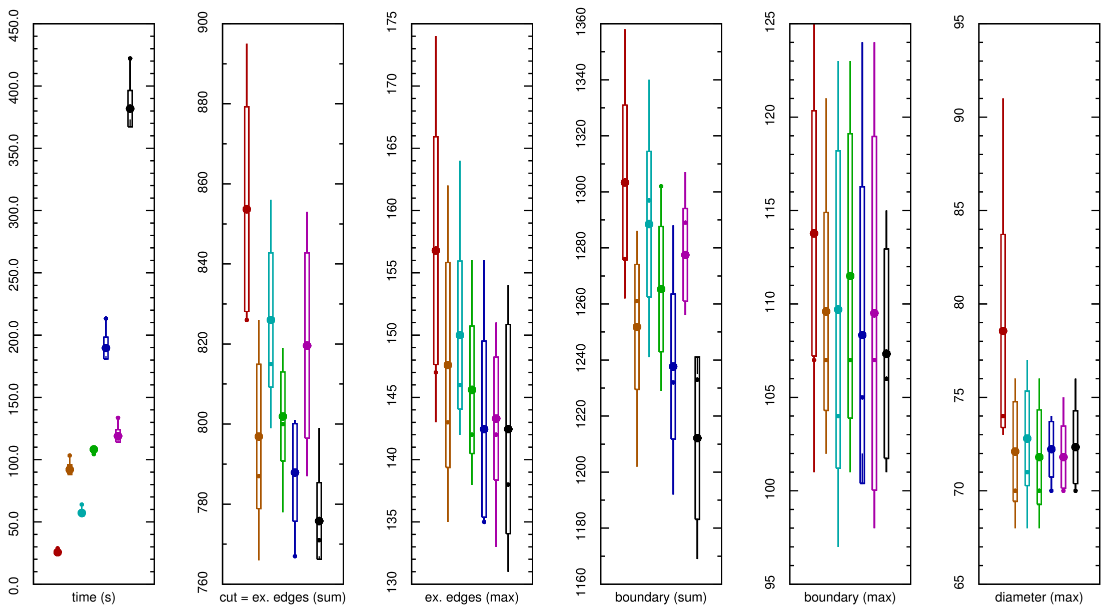


Figure 5.34: Partitioning the 'biplane.9' graph into 16 running 2/2 (red), 2/8 (orange), 4/2 (yellow) 4/4 (green), 4/8 (blue), 8/2 (magenta) and 8/8 (black) iterations/loops with $\phi = 0.001$.

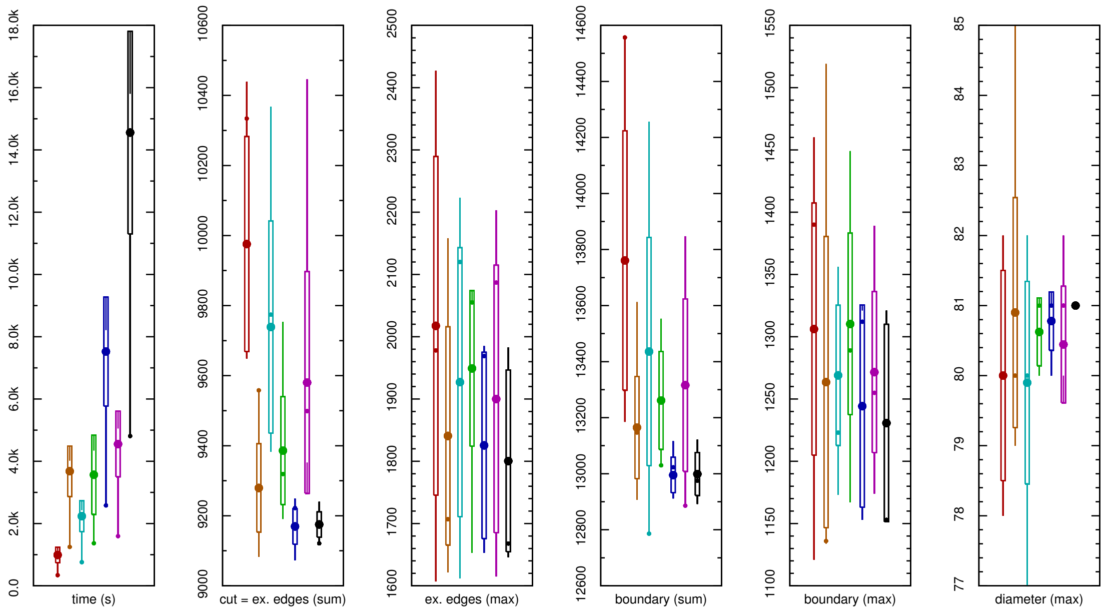


Figure 5.35: Partitioning the 'ocean' graph into 16 parts running 2/2 (red), 2/8 (orange), 4/2 (yellow) 4/4 (green), 4/8 (blue), 8/2 (magenta) and 8/8 (black) iterations/loops with $\phi = 0.001$.

number of boundary vertices as well as the partition diameter.

However, the Grow/Consolidation operation in the optional loops has an even larger impact on the solution quality. As displayed, performing 2 iterations with 8 loops results in better solutions than running 4 iterations with 4 loops. However, the more operations are performed, the smaller is the additional quality gain what also indicates the convergence of the learning process. In many cases, the results of the 4/4 settings are already close to those of the 8/8 ones while the latter requires four times as long. We therefore decide to run 4 iterations with 4 loops per default.

We conclude that the introduction of the optional Grow/Consolidation operation accelerates the learning procedure, because it improves the partition quality quickly. Hence, if the run-time and hence the number of possible operations is limited, it is usually promising to perform more loops in less iterations. Furthermore, we can see that depending on the initial solution, not too many Move/Contractions and Grow/Assignments are required to achieve a good solution quality. In case of the repartitioning problem with an assumed fairly good initial distribution, even one iteration might be sufficient.

Comparison to other Libraries To be able to rate the Flux library, we compare its solutions to those of the state-of-the-art partitioning libraries Metis, Jostle and Party. In the following comparison, Flux runs with its default setting of 4 iterations, 4 loops and $\phi = 0.001$, and uses randomly chosen initial partition centers. Figures 5.36 and 5.37 display the detailed results for the 'biplane.9' and 'ocean' graphs. A summary for the remaining graphs of the test set is listed in table 5.2.

Comparing the run-time, the sequential Flux heuristic is dramatically slower than the other libraries. The difference is about three orders of magnitude. As mentioned, most of the time is spend solving the linear systems of equations. Due to the complexity of the latter, we expect the gap to become even larger with an increasing problem size in case of the sequential implementation.

The second column of the charts shows that all libraries stay inside their allowed balance bounds. kMetis, Party and Flux operate with 3% imbalance allowance, Jostle is invoked with 3%, 1% and 0%, respectively, and pMetis computes completely balanced domains.

Concerning the edge-cut, we confirm the ranking from the experiments in chapter 3. Party finds the solutions with the least cut edges, followed by Jostle and Metis. The results of the Flux library are usually comparable with those of Jostle

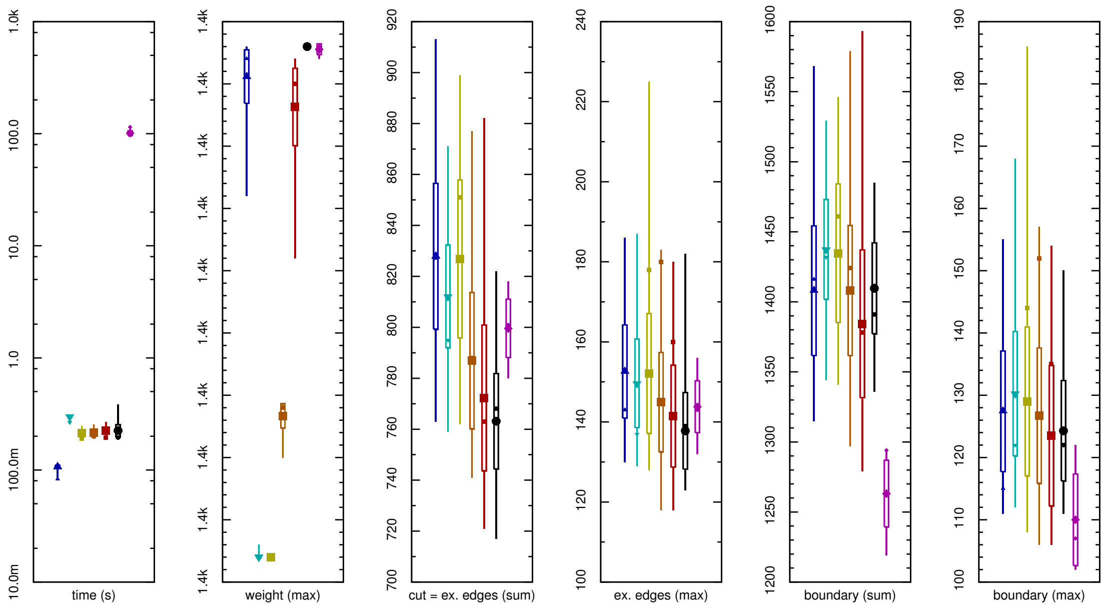


Figure 5.36: Average results for the 'biplane.9' graph divided into 16 parts running kMetis (blue), pMetis (cyan), Jostle (with imbalance 0% (yellow), 1% (orange), 3% (red)), Party (black) and Flux (magenta) with 4 iterations, 4 loops and $\phi = 0.001$.

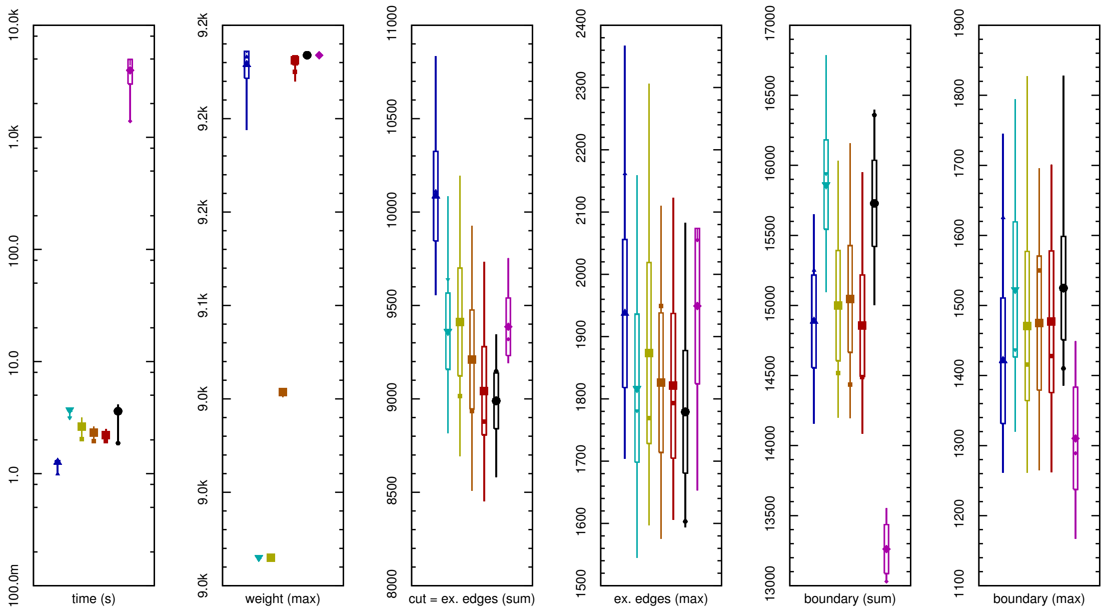


Figure 5.37: Average results for the 'ocean' graph divided into 16 parts using kMetis (blue), pMetis (cyan), Jostle (with imbalance 0% (yellow), 1% (orange), 3% (red)), Party (black) and Flux (magenta) with 4 iterations, 4 loops and $\phi = 0.001$.

Table 5.2: Average partitioning quality when dividing the graphs from the test set into 16 domains with 3% imbalance allowance using kMetis, Jostle, Party and Flux. Flux runs 4 iterations with 4 loops, and set $\phi = 0.001$. The best results are printed bold.

		time	weight	ex. edges		boundary	
graph	library	(s)	max	sum	max	sum	max
airfoil.1	kMetis	0.0	272.9	557.8	102.4	558.1	51.1
	Jostle (3%)	0.0	272.9	545.2	106.1	541.1	51.6
	Party	0.1	274.0	552.7	97.9	552.0	49.3
	Flux	9.4	313.1	**531.5**	**90.8**	**531.7**	**46.0**
biplane.9	kMetis	0.1	1395.6	827.9	152.6	1407.9	127.4
	Jostle (3%)	0.2	1393.1	772.3	141.4	1384.3	123.5
	Party	0.2	1398.0	**763.1**	**137.8**	1409.6	124.3
	Flux	101.3	1397.8	799.6	143.8	**1263.1**	**110.0**
crack	kMetis	0.1	658.2	1251.5	221.8	1229.4	108.1
	Jostle (3%)	0.1	657.8	**1187.1**	**210.5**	**1164.6**	**102.6**
	Party	0.1	660.0	1220.8	215.6	1198.1	105.7
	Flux	34.6	643.0	1241.9	215.9	1218.8	104.7
grid100	kMetis	0.0	641.6	711.0	123.2	1212.0	102.7
	Jostle (3%)	0.1	641.4	659.7	116.1	1205.8	104.0
	Party	0.1	643.8	**614.6**	**105.5**	1165.3	98.8
	Flux	20.3	703.3	662.2	114.2	**1122.7**	**87.6**
ocean	kMetis	1.3	9228.9	10085.0	1936.9	14885.7	1421.1
	Jostle (3%)	2.2	9231.2	9043.2	1821.0	14855.1	1476.7
	Party	3.6	9234.0	**8989.6**	**1779.1**	15728.8	1524.7
	Flux	3967.9	9234.0	9385.9	1949.1	**13261.6**	**1310.2**
stufe.10	kMetis	0.1	1543.3	722.8	125.2	1177.8	101.9
	Jostle (3%)	0.3	1541.7	770.7	167.8	1283.3	139.5
	Party	0.2	1546.4	**678.9**	110.4	1200.0	98.6
	Flux	116.8	1581.4	722.6	**107.1**	**962.5**	**71.1**
wave	kMetis	1.9	10061.9	48660.4	7977.4	25198.8	2071.3
	Jostle (3%)	3.7	10059.2	48411.7	8090.9	25039.4	2078.4
	Party	7.0	10063.0	48621.6	7957.7	24768.6	2038.4
	Flux	6574.8	9969.7	**46416.7**	**7836.5**	**23989.0**	**2010.0**

and better than those computed by pMetis or kMetis. The same holds for the external edges in maximum norm. In some cases like ’airfoil.1’ and ’wave’, Flux can even outperform Party. Note, that all libraries except for Flux focus on a small edge-cut.

In contrast, the Flux heuristic optimizes the domain shape in order to produce small partition boundaries. In almost all tested cases, this goal is achieved and partitionings with the least number of boundary vertices are found in both norms as indicated in table 5.2. In some cases, these numbers are almost 30% smaller than for the next better result, while Flux finds solutions that are about 15% better on average in this metric. An exception is the ’crack’ graph, where the solutions with small edge-cut are almost identical to those with a small boundary.

Graph Repartitioning

In this subsection we present an extraction of the results we obtained when applying the distributed Flux heuristic on the graph sequences of our test set. Again, we report about the influence of Flux’s parameters and evaluate the results of our library with regard to those of the parallel versions of Metis and Jostle.

Parameters Comparing the results of the Flux library with different parameter settings, we come to the same conclusion as in case of the graph partitioning problem. A smaller choice of ϕ and more iterations and loops usually improve the solution quality, but also enlarge the run-time. As an example, figures 5.38 and 5.39 display the results for different parameters for all 101 frames of the ’slowtric’ sequence.

As shown in figure 5.38, reducing ϕ leads to fewer cut-edges, fewer boundary vertices and a smaller partition diameter. While there is a large improvement between 0.05 and 0.005, the gain is smaller when switching to $\phi = 0.0005$. Running 2 loops and 2 iterations results in the worst partition quality, while the 4/4 and 8/8 settings deliver better results as reported in figure 5.39. While a difference between the latter two cases is noticeable if applying the summation norm, it almost vanishes in the maximum norm.

When processing a graph sequence, we record the vertex migration as an additional metric. To be comprehensive, only the sum of outgoing and incoming vertices is displayed. From the data reported in the last row of figures 5.38 and 5.39 we observe that choosing a smaller ϕ or more learning steps increases the number of vertex changes slightly. This is reasonable since a good partition placement with fewer cut edges and smaller boundaries requires more vertices to be migrated.

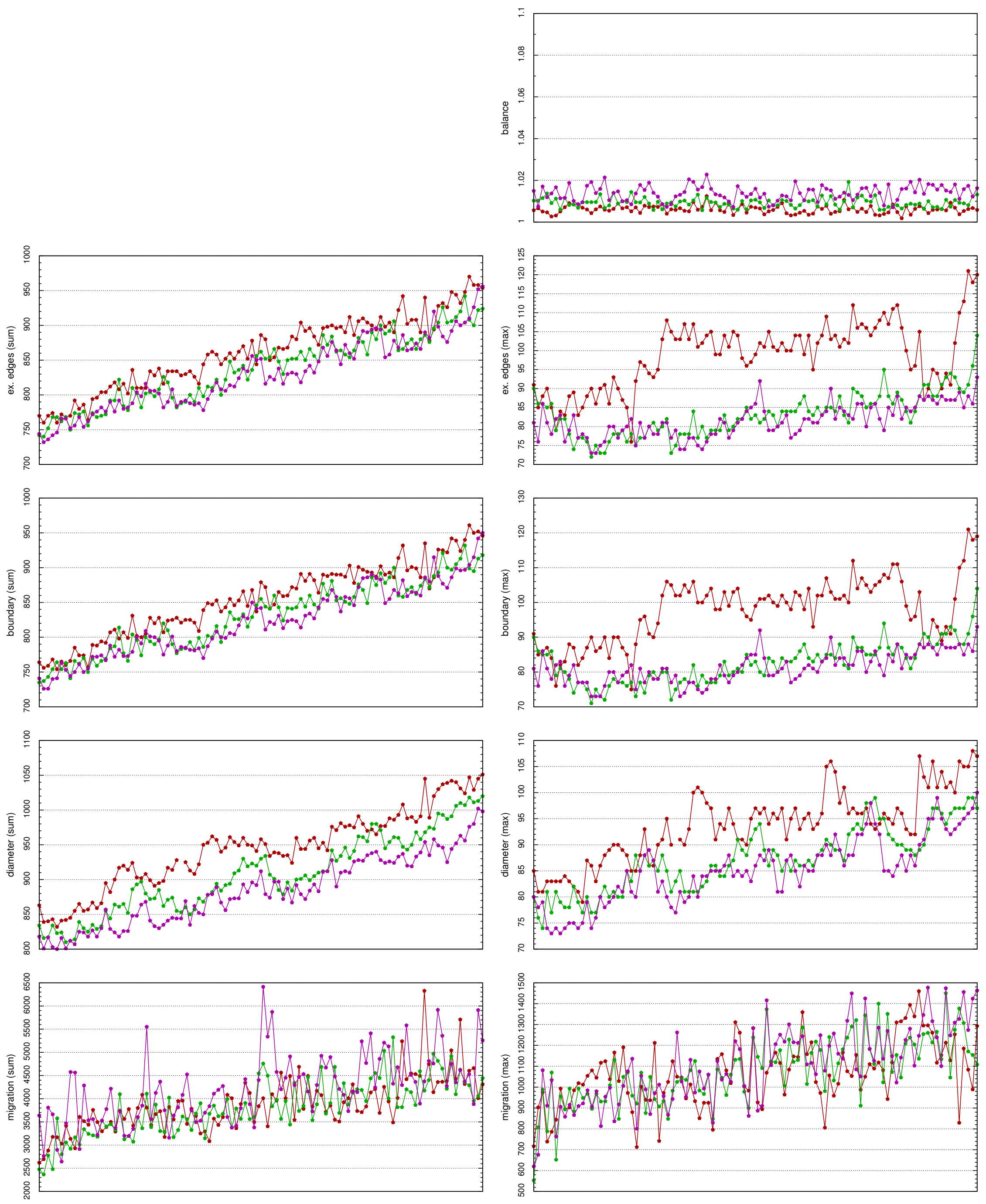


Figure 5.38: Results of the Flux library for all 101 frames of the 'slowtric' benchmark and 12 parts running 4 iterations and 4 loops for different values of ϕ: 0.05 (red), 0.005 (green), 0.0005 (magenta).

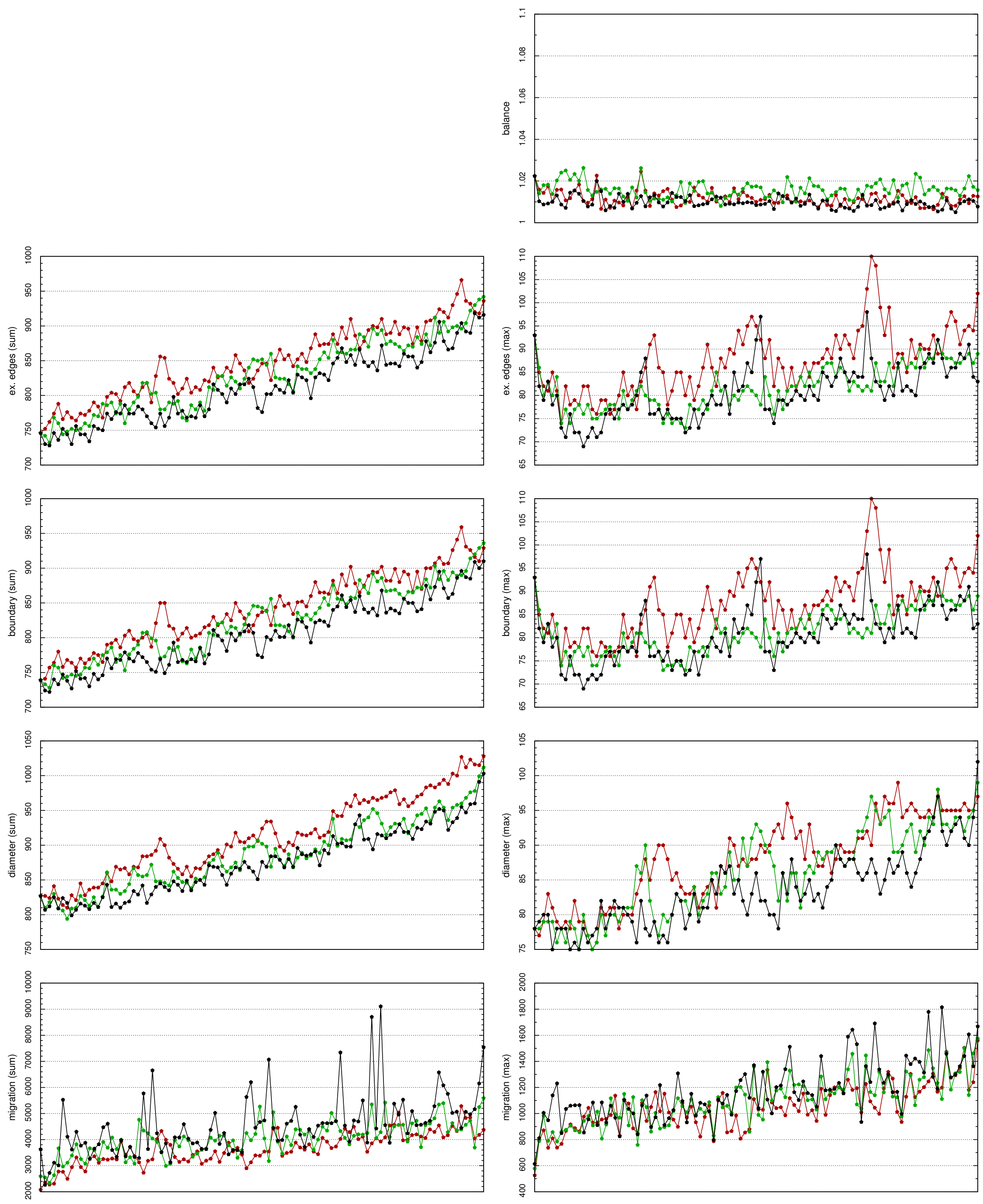


Figure 5.39: Results of the Flux library for all 101 frames of the 'slowtric' benchmark and 12 parts running 2/2 (red) 4/4 (green), and 8/8 (black) iterations/loops, and with the default $\phi = 0.001$.

Comparison to other Libraries The charts listed in figures 5.40 and 5.41 display the solutions of the Flux heuristic together with those of the parallel versions of Metis and Jostle for the 'slowtric' and 'bigslowtric' sequence, respectively. Since the experiments have been performed on different machines, the run-time is not listed, but as before Flux performs by far slowest.

According to the top chart in figure 5.40, the libraries stay inside their balance bounds of 3%, with some exceptions in case of Metis. Furthermore, Jostle ignores the imbalance allowance and almost delivers totally balanced solutions.

The second row contains the number of external edges in summation and maximum norm. Most of the time, Flux is able to find the best solutions while Jostle produces results with an about 5% and Metis with an about 20% higher edge-cut. In the maximum norm, this gap rises to about 12% and 35%, respectively. Similar observations can be made for the boundary displayed in the next row. The number of external edges and boundary vertices seems to be linked and differences are hard to spot for this sequence. It can be explained with the limited vertex degrees (at most 3) in combination with the small graph size.

The diameter shows the same tendencies as the number of external edges or boundary vertices. The advantage of Flux is slightly larger in case of the maximum norm than in the summation norm. Furthermore, there are some frames missing in case of Metis which indicates non-connected partitions.

The last row displays the migration. The results for Metis differs to those of Flux or Jostle, because this library migrates either very few or a large number of vertices. Jostle and Flux proceed much more steadily whereas Flux moves a slightly smaller amount of elements then Jostle.

Since the 'slowtric' benchmark consists of rather small graphs, we use the same geometry description to generate the 'bigslowtric' sequence. It is identical except for the number of vertices, which is increased from around 20000 to 100000. The results of our experiments with this benchmark are displayed in figure 5.41.

The overall picture is very similar to what we observed in case of the 'slowtric' benchmark before. Due to the increased graph size, it is easier to balance the partitions and the Flux library usually finds solutions with less than 0.5% imbalance. As before, Flux computes the partitionings with the fewest cut edges. In the summation norm the advantage to Jostle increases slightly to around 7.5%, and to around 25% when comparing with Metis. In maximum norm, the values of 25% and 55% are even larger. Again, the number of boundary vertices is closely coupled to the number of cut edges, hence a similar improvement can be observed. Looking at the diameter, we see that the partitions computed with Metis are disconnected even more often. As for the 'slowtric' benchmark, Flux determines the domains with the smallest diameter. Concerning the vertex migration, Metis follows its alternative strategy, while Flux moves a very constant number of vertices during the transitions between all frames of the benchmark.

The 'slowtric' and 'bigslowtric' benchmarks indicate that the observations made on the small instances are in general transferable to the larger ones, what we confirm on several other sequences. The benefit of well shaped partitions with straight boundaries becomes larger with an increasing problem size, resulting in a better solution quality in case of Flux compared to the other libraries.

The results for the remaining graph sequences of our test set are similar to the shown 'slowtric' example and endorse the previous observations. To be comprehensive, we only list the values of an arbitrary frame in table 5.3. More specific, we compare the results of Metis, Jostle and Flux after processing frame 66.

As shown, the solution quality is highest when applying the Flux heuristic. In both included norms the domains usually have fewer external edges, the least number of boundary vertices, the smallest diameter. Additionally, a comparable very steady amount of migration occurs. There are only a few exceptions as in case of the 'trace' sequence where our library finds slightly worse results than Jostle. In some cases, Jostle and Flux migrate many more vertices then Metis, but this is only due to the alternative strategy and Metis average value is higher than that of Jostle or Flux.

To provide a better impression of the results, figures 5.42 through 5.45 display the domains computed by Metis, Jostle and Flux in frame 66 for the 'change', 'ring', 'slowtric', and 'trace' benchmarks, respectively. In all examples it is clearly visible that Flux determines a good partition placement that allows well shaped domains with straight boundaries. Jostle's distribution is still reasonable, while, Metis produces many jagged shapes what explains the high number of cut edges and boundary vertices.

Furthermore, we observe that Flux is the only library that produces symmetric looking partitions on symmetric meshes as in case of the 'change' and 'ring' sequence. Tracing the partition placement over several frames, we notice that the domains computed by Flux almost follow the graph geometry. Partitions that are located in refined parts of the mesh move along with these areas. This observation is unique for Flux. In contrast, in the solutions determined by Jostle and Metis the

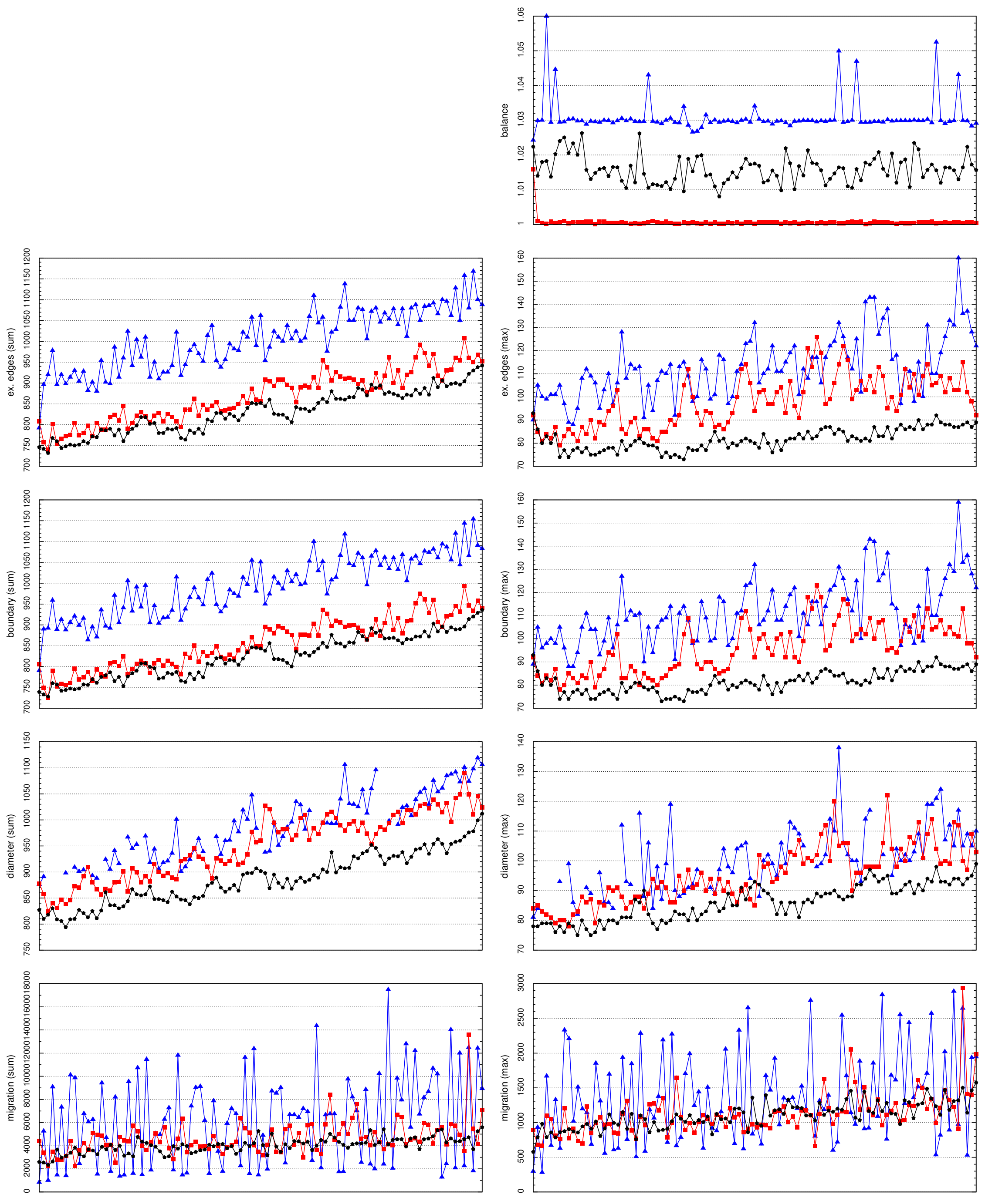


Figure 5.40: Results of all 101 frames of the 'slowtric' benchmark and 12 parts running Metis (blue), Jostle (red), and Flux (black) with 4 iterations, 4 loops, and $\phi = 0.001$.

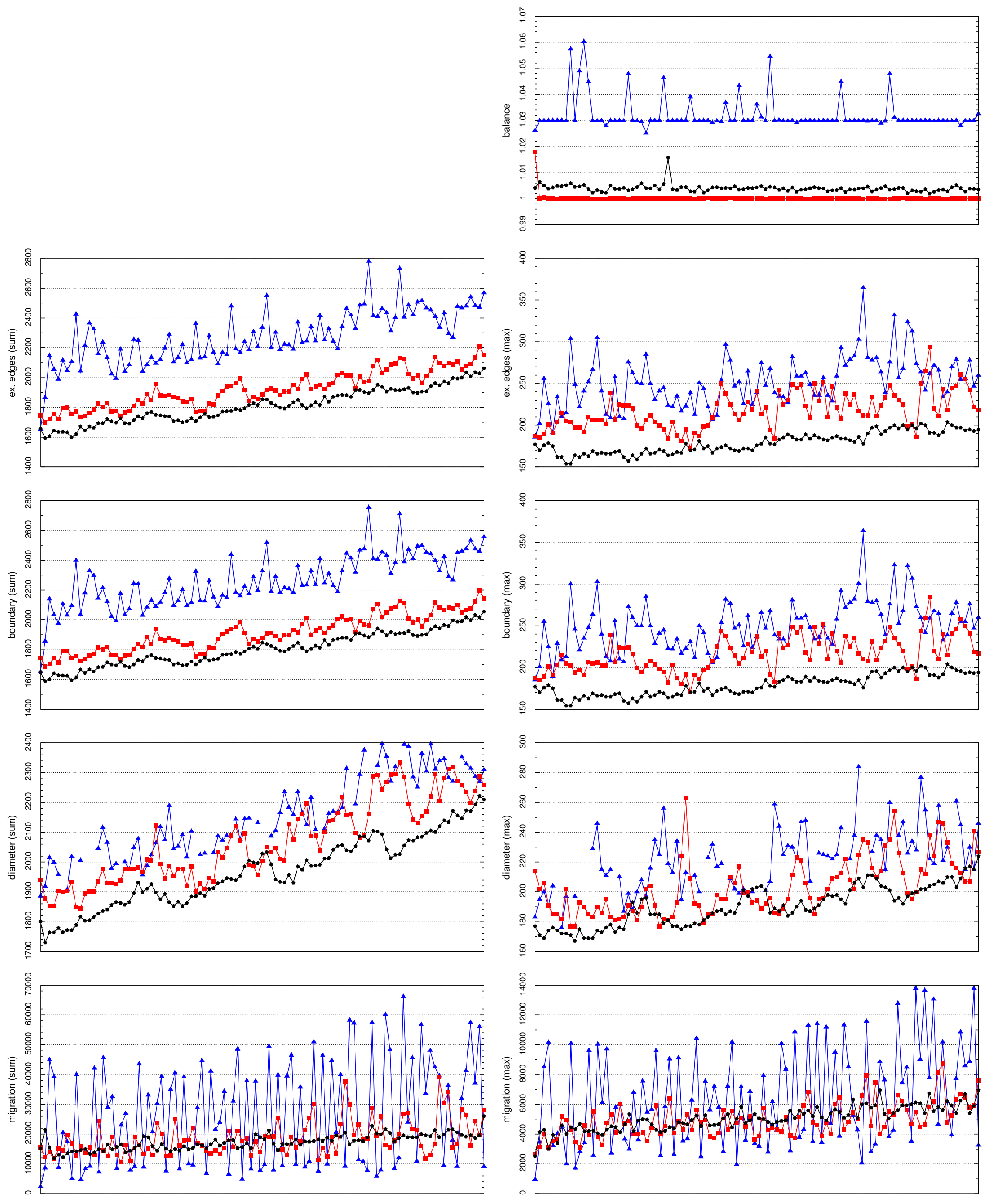


Figure 5.41: Results of all 101 frames of the 'bigslowtric' benchmark and 12 parts running Metis (blue), Jostle (red), and Flux (black) with 4 iterations, 4 loops, and $\phi = 0.001$.

Table 5.3: Solution quality of the 12 partitionings after processing frame 66 with Metis, Jostle and Flux. The best results are printed bold.

		wgt.	ex. edges		boundary		diameter		migration	
seq.	lib.	max	sum	max	sum	max	sum	max	sum	max
	Metis	1.05	57.0	492.0	57.0	492.0	524.0	151.8	**0.0**	**0.0**
refine	Jostle	1.02	49.0	452.0	**48.0**	445.0	492.0	142.4	748.0	244.5
	Flux	1.02	**51.0**	**446.0**	**48.0**	**437.0**	**461.0**	**133.2**	196.0	72.0
	Metis	1.04	79.0	668.0	78.0	652.0	711.0	207.1	**14.0**	**14.0**
change	Jostle	1.00	62.0	558.0	62.0	552.0	674.0	195.3	508.0	171.9
	Flux	1.03	**60.0**	**514.0**	**60.0**	**509.0**	**620.0**	**179.2**	638.0	208.7
	Metis	1.04	50.0	300.0	50.0	291.0			**14.0**	**12.2**
heat	Jostle	1.00	43.0	268.0	43.0	265.0	1008.0	298.5	1254.0	482.9
	Flux	1.02	**40.0**	**262.0**	**39.0**	**261.0**	**989.0**	**294.4**	436.0	155.9
	Metis	1.03	125.0	954.0	125.0	947.0	1299.0	385.9	14026.0	4963.4
ring	Jostle	1.00	119.0	806.0	119.0	798.0	1216.0	365.9	**10714.0**	4031.3
	Flux	1.01	**92.0**	**744.0**	**92.0**	**740.0**	**1146.0**	**346.7**	11694.0	**3879.6**
	Metis	1.03	74.0	554.0	73.0	550.0			8700.0	2820.5
ring2	Jostle	1.00	67.0	482.0	67.0	479.0	888.0	265.1	5740.0	1981.8
	Flux	1.01	**55.0**	**454.0**	**55.0**	**452.0**	**858.0**	**255.4**	**4770.0**	**1649.1**
	Metis	1.03	84.0	760.0	83.0	754.0	794.0	230.5	8096.0	2637.8
circles	Jostle	1.00	83.0	682.0	83.0	664.0	755.0	219.5	**3662.0**	**1449.3**
	Flux	1.01	**77.0**	**642.0**	**77.0**	**639.0**	**685.0**	**198.2**	4714.0	1796.2
	Metis	1.03	106.0	976.0	106.0	974.0	994.0	288.1	6746.0	2367.2
slowtric	Jostle	1.00	119.0	938.0	118.0	927.0	1011.0	295.5	8418.0	2982.9
	Flux	1.02	**86.0**	**854.0**	**86.0**	**847.0**	**900.0**	**260.8**	**5020.0**	**1807.2**
	Metis	1.03	93.0	906.0	93.0	899.0	906.0	265.4	8554.0	3073.8
fasttric	Jostle	1.00	90.0	808.0	90.0	804.0	883.0	257.4	7162.0	2704.7
	Flux	1.02	**75.0**	**792.0**	**75.0**	**783.0**	**818.0**	**237.3**	**7004.0**	**2682.3**
	Metis	1.03	91.0	744.0	88.0	725.0	795.0	231.0	4834.0	1605.8
bubbles	Jostle	1.00	69.0	624.0	**66.0**	614.0	724.0	209.7	2240.0	756.5
	Flux	1.02	**66.0**	**604.0**	**66.0**	**599.0**	**709.0**	**204.9**	**1306.0**	**461.8**
	Metis	1.04	86.0	774.0	84.0	756.0	745.0	217.0	**338.0**	**162.3**
trace	Jostle	1.00	**66.0**	**586.0**	**64.0**	**570.0**	691.0	200.5	1084.0	370.8
	Flux	1.01	70.0	590.0	70.0	585.0	**668.0**	**193.1**	970.0	382.7

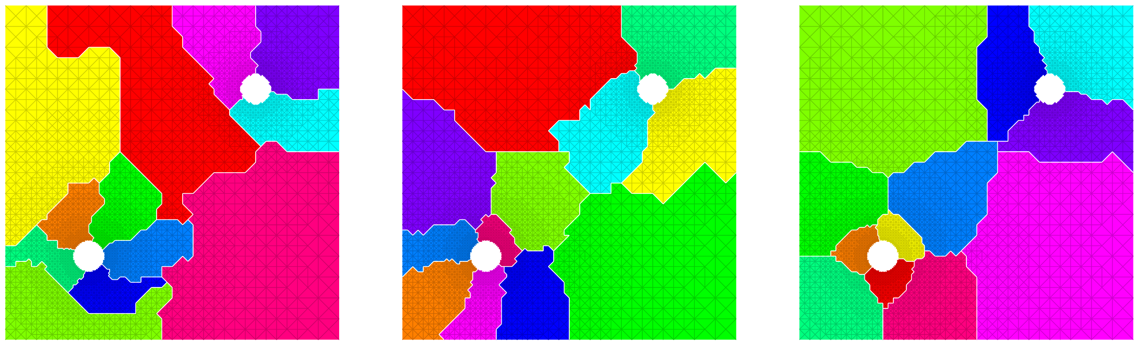


Figure 5.42: Frame 66 of the 'change' benchmark. From left to right the solutions computed by Metis, Jostle and Flux (4 iterations, 4 loops, $\phi = 0.001$) are shown.

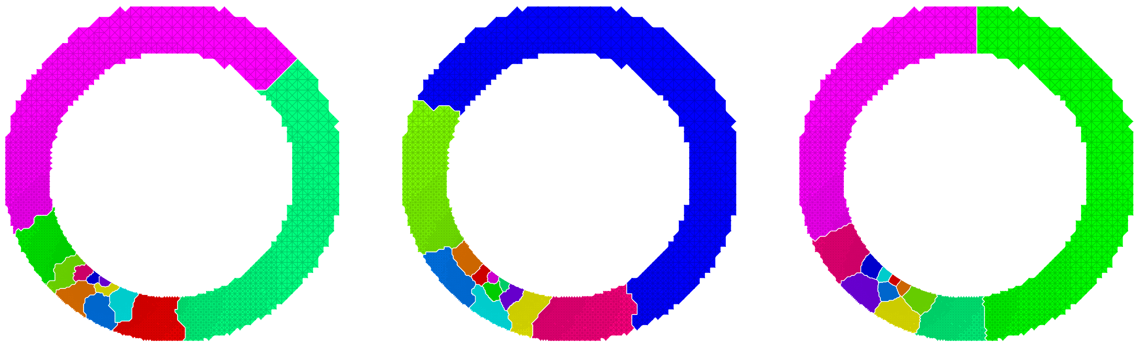


Figure 5.43: Frame 66 of the 'ringrot' benchmark. From left to right the solutions computed by Metis, Jostle and Flux (4 iterations, 4 loops, $\phi = 0.001$) are shown.

geometry moves through the domains. Another point is that the partitions delivered by Flux are always connected. As shown, Metis and less frequently Jostle produce disconnected domains.

Parallel Run-times The previous experiments show that the Flux library outperforms state-of-the-art repartitioning libraries with regard to the solution quality on graphs derived from numerical simulations. However, its run-time is much higher due to the involved numerical computations. In the following we present experimental results that study the parallel execution time in more detail.

In a distributed simulation, each of the processors already contains one part of the mesh when the load balancing algorithm starts. In the experiments we measure the parallel run-times that are required by 8, 16, 32 and 64 processors to repartition the graph. Note that in the current implementation the number of processors must

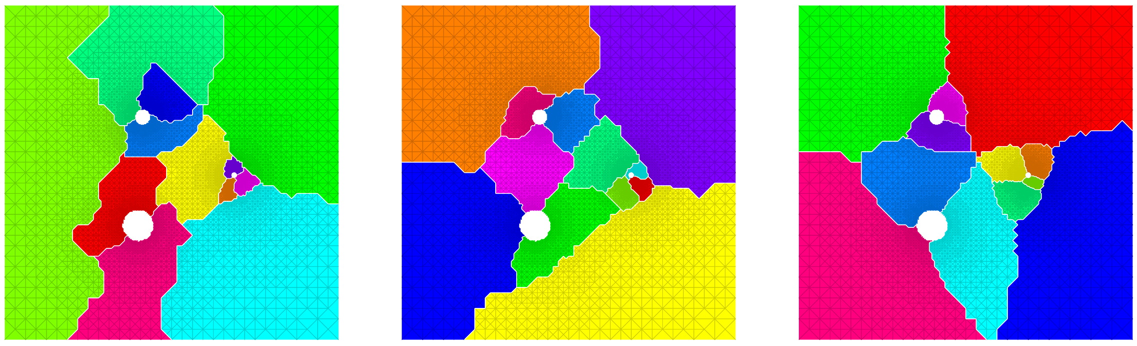


Figure 5.44: Frame 66 of the 'slowtric' benchmark. From left to right the solutions computed by Metis, Jostle and Flux (4 iterations, 4 loops, $\phi = 0.001$) are shown.

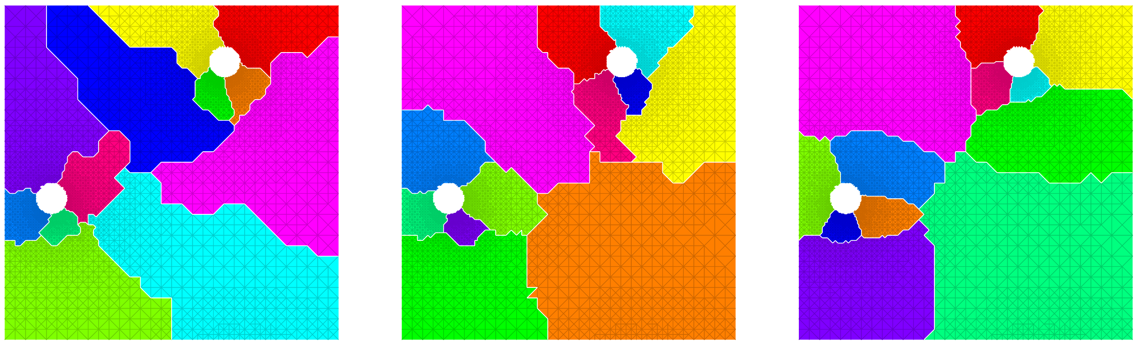


Figure 5.45: Frame 66 of the 'bubble' benchmark. From left to right the solutions computed by Metis, Jostle and Flux (4 iterations, 4 loops, $\phi = 0.001$) are shown.

match the number of domains.

Due to similarities, we limit the presentation to one experiment. The mesh in the included example is taken from the 'bigtrace' benchmark and represents the last transition of that sequence. The considered graph contains 832779 vertices and 1248312 edges. The parameters of our distributed implementation are 2 iterations with 8 loops and $\phi = 0.001$. The experiments are run on the Arminius Cluster, where each of the nodes is equipped with two 64 bit Xeon 3.2 GHz processors and 4 GB of memory. The available interconnection networks are gigabit Ethernet or Infiniband, and communication is realized through the Scampi MPI implementation.

The main difference between the runs are the parameters for the CG solver, the most costly part of the computation. We can choose to apply one (1T) or two (2T) threads per node, precondition with the inverse of the matrix main diagonal (PCG), and decide to disable or enable the partial graph coarsening presented in section 5.6.1. In the latter case, the data of the linear systems can either be distributed via the domain decomposition (DD) or the domain sharing (DS) approach introduced in section 5.6.2. Furthermore, we can choose to communicate via the Ethernet (eth) or Infiniband (ib) interconnection.

The results of the experiment are listed in table 5.4. Besides the run-time, we also list the memory requirements. Furthermore, we include the edge-cut and the total number of boundary vertices for each setting to validate the results. Note that though the same linear system is solved, the numerical imprecision and the different execution order of the processor instructions lead to slightly different results. Furthermore, utilizing the graph hierarchy also influences the solution quality. However, while there are some variations, mainly in case of the edge-cut, no significant changes concerning the solution quality can be reported.

The memory consumption of the Flux library is moderate. Distributing the linear systems according to the domain decomposition scheme, each processing node only stores its own data. In contrast, the domain sharing approach assembles its linear system representing the whole graph. However, it only requires more memory in case of small partition numbers, where a higher amount is allocated during the computation than for the initialization process that loads the graph from disk. This results from the large partition sizes and the small domain distances, meaning that each processor requires the complete information of all other parts. With an increasing partition number, mostly lower hierarchy levels of negligible size are requested.

Comparing the two interconnections networks, we observe that the gigabit Ethernet performs faster when using 8 processing nodes. The reason for this remains unclear, a possible cause are implementation issues in the Scampi library. From 16 partitions on, the Infiniband switch works more efficient. However, this difference is larger when applying the domain decomposition approach, where multiple communications are required in each iteration of the CG solver. Using domain sharing, a large amount of data has to be transfered before and after solving the linear system, but no communication occurs inside the CG solver itself. Connections with higher latency benefit from fewer and larger data packages, what explains the recorded data shown in the table.

Comparing the domain decomposition and the domain sharing approaches themselves, we conclude that domain sharing is always superior. Again, sending only a few large data packages is more efficient than communicating many small ones due to the latency.

Table 5.4: Results of the distributed Flux library repartitioning frame 100 of the 'bigtrace' sequence on the Arminius Cluster in parallel.

P	net		full graph			partly coarsened graph					
			DD			DD			DS		
			CG		PCG	CG	PCG		CG	PCG	
			1T	2T			1T	2T		1T	2T
8	eth	time	296	258	323	892	183	162	630	136	128
	ib	time	398	368	425	1108	242	204	868	181	174
		mem	38	38	42	36	36	36	50	53	53
		cut	2346	2346	2347	2373	2369	2369	2370	2373	2370
		bound	712	712	712	710	708	708	708	704	705
16	eth	time	303	226	327	571	141	131	493	115	115
	ib	time	301	221	322	526	119	98	357	83	75
		mem	36	36	39	35	35	35	35	35	35
		cut	4048	4048	4060	4054	4091	4091	4046	4087	4080
		bound	568	568	568	570	569	569	569	569	569
32	eth	time	440	383	458	423	123	101	293	82	76
	ib	time	423	366	439	395	110	86	265	81	72
		mem	35	35	38	34	34	34	39	40	40
		cut	6210	6210	6201	6262	6226	6226	6283	6219	6247
		bound	425	425	425	415	419	419	415	418	419
64	eth	time	451	401	464	227	81	76	130	54	49
	ib	time	426	378	449	189	66	63	132	49	47
		mem	35	35	38	34	34	34	34	34	34
		cut	9256	9256	9256	9229	9251	9251	9247	9259	9271
		bound	317	317	317	317	321	321	320	321	321

The implemented CG solver can optionally precondition the system with the inverse of the matrix diagonal. Since the input graph is unweighted and the dual graph of the mesh is almost 3-regular, a benefit can only be expected when using different hierarchy levels. Accordingly, when solving the linear systems on the full graph only, the run-times are shorter when disabling the preconditioner as shown in the first three columns. However, computing a solution on the partly coarsened graphs, preconditioning reduces the run-time by a factor between 3 and 5 as reported in the three middle and right three columns.

From the table we can also see that running a separate thread on each of the processors further speeds up the execution, but the run-time reduction is very moderate. We think that the limited benefit might result from interferences with the Scampi library.

So far we have looked into values that are computed with the same number of processing nodes. For the following vertical comparison, one should have in mind that the number of linear systems that have to be solved equals the number of domains. Hence, a linear speedup results in a constant execution time.

When computing solutions on the full graph, more partitions lead to a longer run-time. The increase is larger in case of the gigabit Ethernet, which indicates that this behavior occurs not only because of the computational load but also due to the growing communication volume. However, in case of the partly coarsened approach the opposite can be observed and a larger partition number requires less execution time. This can be explained with the reduction of the linear system size which occurs because more and more parts can be replaced with lower levels of the hierarchy. However, since the graph size is constant, the partitions become smaller. Hence, less local computations have to be executed and the fraction of the communication overhead rises, what slows down and eventually limits the benefit. Hence, we suppose that Flux scales best if the problem size grows together with the number of processing nodes.

5.7.4 Application in PadFEM

To test the Flux library in a real application, we integrate it into the PadFEM [BM05] package. This package includes a variety of tools to solve partial differential equations on tetrahedral meshes in parallel. In regions where the solution quality does not meet the requirements, it generates refinements through its distributed adaptation module. For the graph partitioning process, every tetrahedron of the mesh is associated with a vertex of the dual graph, which then forms the input data for the partitioning and repartitioning algorithms. Hence, the dynamic changes in the mesh result in graph sequences.

From our experiments we conclude that Flux also works reliable on graphs derived from 3-dimensional simulations. As in case of the 2-dimensional sequences, it computes well shaped connected domains with few boundary vertices. Figure 5.46 gives an example of a 3-dimensional mesh occurring in the DFG standard simulation benchmark [MS96] after several refinement steps. The domains are extruded for a better view. Since the initial distribution has been computed with Party, there

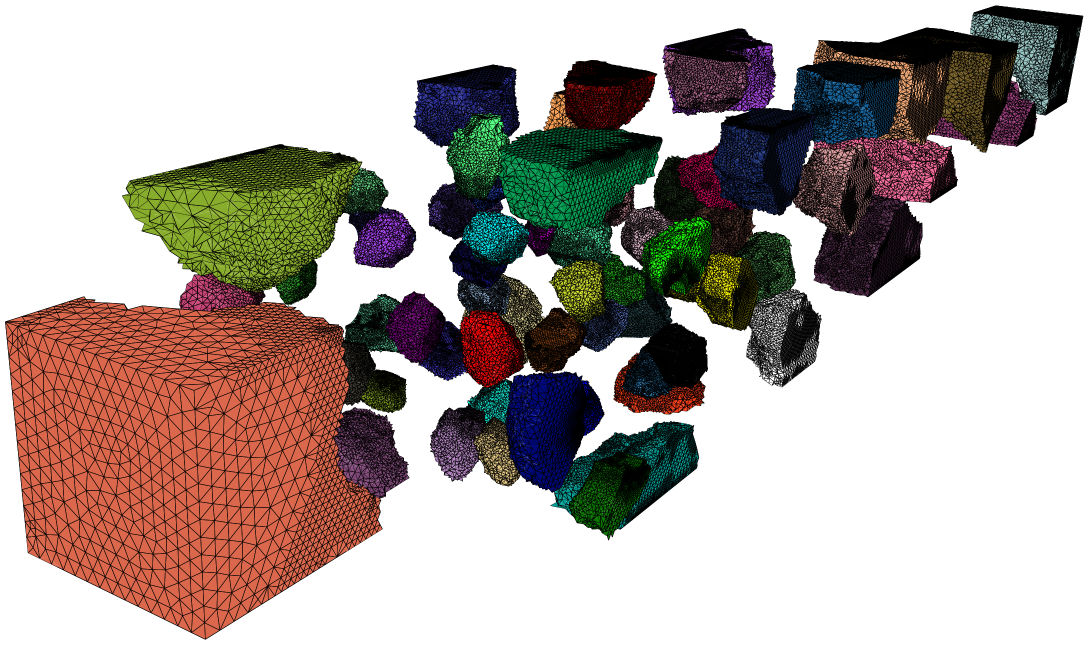


Figure 5.46: Extruded 64 partitioning during a 3-dimensional parallel adaptive simulation.

are still some areas that contain square like domains. However, these are slowly transformed by Flux into a ball like shape. In the refined regions, this has already happened. Though Flux performs slowly in comparison to other heuristics, its fraction of the overall run-time is only around 2% in case of the perfomed fluid flow computation.

Unfortunately, we cannot directly compare the results of Flux with those of Metis or Jostle. Metis produces disconnected partitions, which form an illegal input for some of the mathematical computations. Jostle crashes once the graph size becomes to large. Hence, the Flux library is currently the only working load balancer. However, we think that the properties observed in the 2-dimensional test-cases also hold for input from 3-dimensional meshes. Remember that Flux does not use vertex coordinates but only relies on the connectivity information of the dual graph. The degree of 3-dimensional dual graphs is usually higher than for two dimensions, hence a larger number of domains is required to benefit from the partly coarsened hierarchy levels. Again, Flux performs much slower than libraries that base on the multi-level scheme, but its solution quality is superior. Depending on the application, this can either be beneficial or not.

5.7.5 Upshot

The presented new graph partitioning and repartitioning heuristic Flux delivers high quality solutions. Although it does not contain any explicit objectives other than the balancing, it is able to compute connected well shaped domains with short and straight boundaries, a small diameter and little cut edges. Hence it very well satisfies the demands of a range of applications, like the given example of balancing parallel adaptive numerical simulations.

We impute the superior solution quality to the diffusion process that gathers global information about the graph structure, screens relevant information due to the properties of the load distribution, and directs the learning framework in order to find good partition placements. Heuristics that are implemented in Metis, Jostle and Party are missing this kind of information. Hence, although these libraries contain very good refinement algorithms, only local changes cannot eliminate the effects of wrong global conditions.

Furthermore, the proposed shape optimizing algorithm mainly consists of easily parallelizable computations, what might an important advantage over commonly used vertex exchange refinement heuristics, considering the upcoming multi-core processors. The drawback of the presented approach is the high computational costs, which can be reduced slightly by the described techniques.

6 Conclusion

In this thesis we address the graph partitioning and repartitioning problem. In particular, we focus on load balancing in a distributed numerical simulation. We provide an overview on commonly applied techniques and list state-of-the-art implementations. Furthermore, we present several enhancements to existing approaches. We introduce the Graph-Filling Curves, present a better evaluation scheme, and improve the Helpful-Set bisection refinement procedure that is implemented in the Party library. Additionally, we summarize some new results on diffusion schemes for certain network types.

The main contribution of this thesis is the new repartitioning heuristic Flux. In contrast to other existing libraries, we follow an alternative strategy and do not focus on a small edge-cut. Instead, we optimize the domain shapes by applying a diffusion process embedded inside a learning framework. We formulate and analyse the new diffusion scheme FOS/C that meets our special requirements.

While the solutions computed with Flux also contain a small number of cut edges, our experiments show that the main benefit of the new heuristic is the ability to find well shaped and connected domains with a small number of boundary vertices, what meets the requirements of distributed numerical simulations. Furthermore, the proposed mechanism contains many calculations that can easily be performed in parallel. The main drawback of Flux is the large number of involved numerical computations. We present some techniques that reduce the computational load. Nevertheless, in its current implementation the Flux library is two to three magnitudes slower than other distributed state-of-the-art heuristics. Depending on the application, this overhead might or might not be justified by the better solution quality. Since some of the algorithms in the PadFEM simulation environment depend on good partition shapes and require connected domains, the Flux library is currently the only applicable load balancer.

For future prospects, we think that a combination of a local refinement strategy together with a technique that collects some global information beforehand is most promising. Although local vertex exchanges can quickly improve the solution quality, the large number of equally rated improvement steps often causes wrong decisions that downgrade the overall situation. Some additional infomation about the structural properties in the for a partition relvant regions of the graph can guide the exchange procedure, and differentiate between the otherwise indistinguishable local choices. Hence, combining local and global methods might result in both, a short run-time and high quality solutions.

Bibliography

[Avi83] D. Avis. A survey of heuristics for the weighted matching problem. *Networks*, 13:475–493, 1983.

[BCLS87] T.N. Bui, S. Chaudhuri, F.T. Leighton, and M. Sisper. Graph bisection algorithms with good average case behaviour. *Combinatorica*, 7(2):171–191, 1987.

[BEM$^+$04] S. Bezrukov, R. Elsässer, B. Monien, R. Preis, and J.-P. Tillich. New spectral lower bounds on the bisection width of graphs. *Theoretical Computer Science*, 320:155–174, 2004.

[BM05] S. Blazy and O. Marquardt. Parallel finite element computations of three-dimensional benchmark problems. In M. Kowarschik F. Hülsemann and U. Rüde, editors, *Proc. of the 18th Symposium on Simulationstechniques (ASIM2005)*, pages 152–157, 2005.

[BO98] R. Biswas and L. Oliker. PLUM: Parallel load balancing for adaptive unstructured meshes. *Parallel and Distributed Computing*, 51(2):150–177, 1998.

[Boi90] J.E. Boillat. Load balancing and poisson equation in a graph. *Concurrency - Practice & Experience*, 2:289–313, 1990.

[Bol88] B. Bollobas. The isoperimetric number of random regular graphs. *European Journal of Combinatorics*, 9:241–244, 1988.

[CDS95] D.M. Cvetkovic, M. Doob, and H. Sachs. *Spectra of Graphs*. Johann Ambrosius Barth, 3rd edition, 1995.

[CE88] L.H. Clark and R.C. Entringer. The bisection width of cubic graphs. *Bulletin of th Australian Mathematical Society*, 39:389–396, 1988.

[Chi92] P. Chiu. Cubic ramanujan graphs. *Combinatorica*, 12(3):275–285, 1992.

[Cyb89] G. Cybenko. Load balancing for distributed memory multiprocessors. *Parallel and Distributed Computing*, 7:279–301, 1989.

[DFM98] R. Diekmann, A. Frommer, and B. Monien. Nearest neighbor load balancing on graphs. In *Proceedings of the European Symposium on Algorithms (ESA'98)*, volume 1461 of *LNCS*, pages 429–440, 1998.

[DFM99] R. Diekmann, A. Frommer, and B. Monien. Efficient schemes for nearest neighbor load balancing. *Parallel Computing*, 25(7):789–812, 1999.

[DH03a] D.E. Drake and S. Hougardy. Improved linear time approximation algorithms for weighted matchings. In *Approximation, Randomization, and Combinatorial Optimization (APPROX/RANDOM 03)*, LNCS 2764, pages 14–23, 2003.

[DH03b] D.E. Drake and S. Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85(4):211–213, 2003.

[DHB02] S.K. Das, D.J. Harvey, and R. Biswas. MinEX: a latency-tolerant dynamic partitioner for grid computing applications. *Future Generation Computing Systems*, 18(4):477–489, 2002.

[DMN97] R. Diekmann, S. Muthukrishnan, and M.V. Nayakkankuppam. Engineering diffusive load balancing algorithms using experiments. In G. Bilardi, A. Ferreira, R. Lüling, and J. Rolim, editors, *Solving Irregulary Structured Problems in Parallel (IRREGULAR)*, LNCS 1253, pages 111–122, 1997.

[DMP95] R. Diekmann, B. Monien, and R. Preis. Using helpful sets to improve graph bisections. In D.F. Hsu, A.L. Rosenberg, and D. Sotteau, editors, *Interconnection Networks and Mapping and Scheduling Parallel Computations*, volume 21 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 57–73. AMS, 1995.

[DPS02] J. Diaz, J. Petit, and M. Serna. A survey of graph layout problems. *ACM Comput. Surv.*, 34(3):313–356, 2002.

[DPSW00] R. Diekmann, R. Preis, F. Schlimbach, and C. Walshaw. Shape-optimized mesh partitioning and load balancing for parallel adaptive FEM. *Journal of Parallel Computing*, 26:1555–1581, 2000.

[Dut93] S. Dutt. New faster kernighan-lin-type graph-partitioning algorithms. In *Proceedings of IEEE/ACM International Conference on CAD*, pages 370–377, 1993.

[EFMP99] R. Elsässer, A. Frommer, B. Monien, and R. Preis. Optimal and alternating-direction loadbalancing schemes. In P. Amestoy et al., editor, *EuroPar Parallel Processing*, LNCS 1685, pages 280–290, 1999.

[EKM03] R. Elsässer, R. Kralovic, and B. Monien. Sparse topologies with small spectrum size. *Theor. Comput. Sci.*, 307(3):549–565, 2003.

[Els02] U. Elsner. *Static and Dynamic Graph Partitioning. A comparative study of existing algorithms.* Logos, Berlin, 2002.

[EMP02] R. Elsässer, B. Monien, and R. Preis. Diffusion schemes for load balancing on heterogeneous networks. *Theory of Computing Systems*, 35:305–320, 2002.

[EMS02] R. Elsässer, B. Monien, and S. Schamberger. Toward optimal diffusion matrices. In *Proceedings of the 16th International Parallel and Distributed Processing Symposium (IPDPS'02)*, page 67 (CD). IEEE Computer Society, 2002.

[EMS04] R. Elsässer, B. Monien, and S. Schamberger. Load balancing in dynamic networks. In *Proceedings of the 7th international Symposium on Parallel Architectures, Algorithms and Networks, (I-SPAN'04)*, pages 193–200. IEEE Conputer Society, 2004.

[Fie73] M. Fiedler. Algebraic connectivity of graphs. *Czechoslovak Mathematical Journal, Praha*, 23(98):298–305, 1973.

[Fie75] M. Fiedler. A property of eigenvectors of nonnegative symmetric matrices and its application to graph theory. *Czechoslovak Mathematical Journal, Praha*, 25(100):619–633, 1975.

[FK02] U. Feige and R. Krauthgamer. A polylogarithmic approximation of the minimum bisection. *SIAM Journal on Computing*, 31(4):1090–1118, 2002.

[FKN00] U. Feige, R. Krauthgamer, and K. Nissim. Approximating the minimum bisection size. In *Proceedings of the Symposium on Theory of Computing (STOC'00)*, pages 530–536, 2000.

[FM82] C.M. Fiduccia and R.M. Mattheyses. A linear-time heuristic for improving network partitions. In *Proceedings of IEEE Design Automation Conference*, pages 175–181, 1982.

[GJ79] M.R. Garey and D.S. Johnson. *COMPUTERS AND INTRACTABILITY - A Guide to the Theory of NP-Completeness.* Freemann, 1979.

[GMS98] B. Ghosh, S. Muthukrishnan, and M.H. Schultz. First- and second-order diffusive methods for rapid, coarse, distributed load balancing. *Theory of Computing Systems*, 31(4):331–354, 1998.

[Gup96] A. Gupta. WGPP: Watson graph partitioning package. Technical Report RC 20453, IBM Research Report, 1996.

[HB99] Y.F. Hu and R.F. Blake. An improved diffusion algorithm for dynamic load balancing. *Parallel Computing*, 25(4):417–444, 1999.

[HD00] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanics and Engineering*, 184:485–500, 2000.

[Hen98] B. Hendrickson. Graph partitioning and parallel solvers: Has the emperor no clothes? In *Irregular'98*, number 1457 in LNCS, pages 218–225, 1998.

[Hil91] D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

[HL93] B. Hendrickson and R.W. Leland. Multidimensional spectral load balancing. In *Proceedings of the 6th SIAM Conference on Parallel Processing*, pages 953–961, 1993.

[HL94] B. Hendrickson and R.W. Leland. *The Chaco user's guide – Version 2.0*, 1994.

[HL95] B. Hendrickson and R.W. Leland. A multi-level algorithm for partitioning graphs. In *Supercomputing*, 1995.

[HM91] J. Hromkovic and B. Monien. The bisection problem for graphs of degree 4. In *Proceedings of Mathematical Foundations of Computer Science (MFCS '91)*, volume 520 of *LNCS*, pages 211–220, 1991.

[JM96] D.B. Johnson and D.A. Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[Kaa88] E.F. Kaasschieter. Preconditioned conjugate gradients for solving singular systems. *Journal of Computational and Applied Mathematics*, 24(1-2):265–275, 1988.

[KK96] G. Karypis and V. Kumar. Parallel multilevel graph partitioning. In *IPPS*, pages 314–319, 1996.

[KK97] G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph-partitioning algorithm. In *Proc. 8th SIAM Conf. on Parallel Processing for Scientific Computing*, 1997.

[KK98a] G. Karypis and V. Kumar. *MeTis: A Software Package for Partitioning Unstrctured Graphs, Partitioning Meshes, [...], Version 4.0*, 1998.

[KK98b] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1), 1998.

[KK99] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1), 1999.

[KL70] B.W. Kernighan and S. Lin. An effective heuristic procedure for partitioning graphs. *The Bell Systems Technical Journal*, pages 291–307, 1970.

[KM92] A.V. Kostochka and L.S. Melnikov. On bounds of the bisection width of cubic graphs. In J. Nesetril and M. Fiedler, editors, *Proc. Fourth Czechoslovakian Symp. on Combinatorics, Graphs and Complexity*, pages 151–154. Elsevier Science Publishers B.V., 1992.

[KM93] A.V. Kostochka and L.S. Melnikov. On a lower bound for the isoperimetric number of cubic graphs. In V.F. Kolchin et al., editor, *Probabilistic Methods in Discrete Mathematics, Proceedings of the 3rd International Petrozavodsk Conference*, volume 1 of *Progress in Pure and Applied Discrete Mathematics*, pages 251–265. TVP/VSP, 1993.

[Kri84] B. Krishnamurthy. An improved min-cut algorithm for partitioning vlsi networks. *IEEE Trans. Computers*, 33(5):438–446, 1984.

[Lei92] F.T. Leighton. *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes.* Morgan Kaufmann Publishers, 1992.

[Mac67] J.B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. Berkeley, University of California Press, 1967.

[MD97] B. Monien and R. Diekmann. A local graph partitioning heuristic meeting bisection bounds. In *SIAM Conference on Parallel Processing for Scientific Computing*, 1997.

[MMS06] H. Meyerhenke, B. Monien, and S. Schamberger. Accelerating shape optimizing load balancing for parallel FEM simulations by algebraic multigrid. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium, (IPDPS'06)*, page 57 (CD). IEEE Computer Society, 2006.

[Mor94] M. Morgenstern. Existence and explicit constructions of $q+1$ regular ramanujan graphs for every prime power q. *Journal of Combinatorial Theory, Series B*, 62(1):44–62, 1994.

[MP01] B. Monien and R. Preis. Upper bounds on the bisection width of 3- and 4-regular graphs. In *26th International Symposium on Mathematical Foundations of Computer Science (MFCS)*, LNCS 2136, pages 524–536, 2001.

[MPD00] B. Monien, R. Preis, and R. Diekmann. Quality matching and local improvement for multilevel graph-partitioning. *Parallel Computing*, 26(12):1609–1634, 2000.

[MS96] S. Turek M. Schaefer. Benchmark computations of laminar flow around cylinder. In *Flow Simulation with High-Performance Computers II*, volume 52 of *Notes on Numerical Fluid Mechanics*, pages 547–566, 1996.

[MS04] B. Monien and S. Schambeger. Graph partitioning with the party library: Helpful-sets in practice. In *Proceedings of the 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 198–205. IEEE Computer Society Press, 2004.

[MS05] O. Marquardt and S. Schamberger. Open benchmarks for load balancing heuristics in parallel adaptive finite element computations. In *Proc. Intern. Conf. on Par. and Dist. Processing Techniques and Appl., (PDPTA'05)*. CSREA Press, 2005.

[Pre98] R. Preis. The PARTY graphpartitioning-library – user manual, 1998.

[Pre99] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *Symposium on Theoretical Aspects of Computer Science (STACS'99)*, volume 1563 of *LNCS*, pages 259–269, 1999.

[Pre00] R. Preis. Analyses and design of efficient graph partitioning methods, 2000. Dissertation, Universität Paderborn, Germany.

[PSL90] A. Pothen, H.D. Simon, and K.P. Liu. Partitioning sparse matrices with eigenvectors of graphs. *SIAM Journal on Matrix Analysis and Applications*, 11(3):430–452, 1990.

[Sag94] H. Sagan. *Space Filling Curves.* Springer, 1994.

[Sch04] S. Schamberger. On partitioning FEM graphs using diffusion. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium, (IPDPS'04)*, page 277 (CD). IEEE Computer Society, 2004.

[Sch05] S. Schamberger. A shape optimizing load distribution heuristic for parallel adaptive FEM computations. In *Proceedings of the 8th International Conference on Parallel Computing Technologies*, volume 3606 of *LNCS*, pages 263–277, 2005.

[SG99] V.S. Sunderarm and G.A. Geist. Heterogeneous parallel and distributed computing. *Parallel Computing*, 25:1699–1721, 1999.

[SKK97] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel Distributed Computing*, 47(2):109–124, 1997.

[SKK00] K. Schloegel, G. Karypis, and V. Kumar. A unified algorithm for load-balancing adaptive scientific simulations. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing*, page 59 (CD). IEEE Computer Society, 2000.

[SKK02] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high performance scientific simulations. In *The Sourcebook of Parallel Computing.* Morgan Kaufmann, 2002.

[SW03] S. Schamberger and J. M. Wierum. Graph partitioning in scientific simulations: Multilevel schemes vs. space-filling curves. In *Parallel Computing Technologies, PACT'03*, number 2763 in LNCS, pages 165–179, 2003.

[Wal] C. Walshaw. The graph partitioning archive. http://staffweb.cms.gre.ac.uk/c.walshaw/partition/.

[Wal00] C. Walshaw. *The Jostle user manual: Version 2.2.* University of Greenwich, 2000.

[Wal02] C. Walshaw. *The parallel JOSTLE library user guide: Version 3.0*, 2002.

[WC00] C. Walshaw and M. Cross. Mesh partitioning: A multilevel balancing and refinement algorithm. *SIAM Journal on Scientific Computing*, 22(1):63–80, 2000.

[WCE97] C. Walshaw, M. Cross, and M.G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *Journal on Parallel Distributed Computing*, 47(2):102–108, 1997.

[WL02] Y. Wang and X.-Y. Li. Geometric spanners for wireless ad hoc networks. In *22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 171–180. IEEE, 2002.

[XL97] C. Xu and F.C.M. Lau. *Load Balancing in Parallel Computers.* Kluwer, 1997.