Load Balancing Strategies for Distributed Memory Machines *

Ralf Diekmann and Burkhard Monien

Department of Computer Science
University of Paderborn
Fürstenallee 11,
D-33102 Paderborn, Germany

and

Robert Preis

Heinz Nixdorf Institut
University of Paderborn
Fürstenallee 11,
D-33102 Paderborn, Germany

Email: {diek, bm, robsy}@uni-paderborn.de
WWW: http://www.uni-paderborn.de/cs/ag-monien/

*This work was supported by the DFG-Sonderforschungsbereich 376 "Massive Parallelität: Algorithmen, Entwurfsmethoden, Anwendungen", the DFG-Graduiertenkolleg "Parallele Rechnernetze in der Produktionstechnik" at the Heinz Nixdorf Institut, and the EC ESPRIT Long Term Research Project No. 20244 (ALCOM-IT).

*Parts of this work appeared in Karsch/Monien/Satz (ed.): *Multi-Scale Phenomena and their Simulation*, World Scientific, 255-266, 1997.

Abstract

Load balancing in large parallel systems with distributed memory is a difficult task often influencing the overall efficiency of applications substantially. A number of efficient distributed load balancing strategies have been developed in the recent years. Although they are currently not generally available as part of parallel operating systems, it is often not difficult to integrate them into applications.

In this paper we use the notion of *graph embedding* to develop a general description for different kinds of load balancing problems. Based on the characteristics of the application and the parallel system we classify the problems into five groups. For the case of applications from the field of scientific computing, useful methods are described in more detail.

If an application is static, the load balancing problem reduces to the task of mapping the application graph onto the processor network. For modern parallel architectures, this problem can further be relaxed to graph partitioning. We give a survey of the most important graph partitioning heuristics.

If the application is dynamic, e.g. adaptively refining meshes in finite element calculations, a number of different dynamic load balancing strategies can be used to handle the varying utilization of processors. We describe several methods based on a two-step approach: 1.) Considering how much load to move in which direction and 2.) which load to move.

1 Introduction

The use of parallel processing is established in nearly all areas of science and technology. Small symmetric multiprocessor systems (SMPs) can be found within many of the powerful modern workstations. Medium-sized machines with (virtually) shared memory are becoming more and more common. While the techniques for parallelizing application algorithms are generally available, most of them fail if scalable codes for large, massively parallel systems have to be designed. The principle of globally shared memory is physically not scalable to more than a small number of processors. Larger systems typically consist of processors with own local memory, i.e. they implement *distributed memory*. Data exchange in these machines use *message passing* via a communication network.

The problem of load balancing becomes much more difficult in large distributed systems. Algorithms have to minimize both load imbalance and communication overhead of the application. Additionally, they should be efficient themselves (i.e. they should balance the load with as little overhead as possible) and they should be scalable.

For a number of different applications, efficient distributed load balancing strategies have been developed in the recent years [53]. Although they are currently not generally available, at least not as part of parallel operating systems, it is often not difficult to integrate them into applications [51, 86]. However, the algorithms to be used for load balancing still depend heavily on the characteristics of the application.

Aim of this paper is to give a survey and classification of different load balancing scenarios based on application characteristics and communication capabilities of parallel systems. Especially for the case of applications from the field of scientific computing, useful methods for both, static and dynamic load balancing are described in more detail.

In Section 2 we discuss different models of parallel systems. Using the *Distributed Memory Model (DMM)*, load balancing can be expressed as graph embedding problem. Based on this modeling, we classify different load balancing scenarios into static and dynamic problems. The dynamic ones are split again into four groups according to the communication features of the parallel system and the characteristics of the application. Properties and possible balancing methods for three of the four dynamic classes are discussed briefly.

If the application is static, the task of load balancing reduces to a mapping problem which can be further relaxed to a graph partitioning problem. In Section 3 we give a survey of the most important graph partitioning heuristics. We describe the algorithms in detail and compare results of several partitioning libraries implementing different heuristics.

Section 4 deals with load balancing for dynamic applications from the field of scientific computing such as adaptive finite element simulations, which refers to the fourth dynamic class distinguished in Section 2. Problems and possible solutions are discussed in detail.

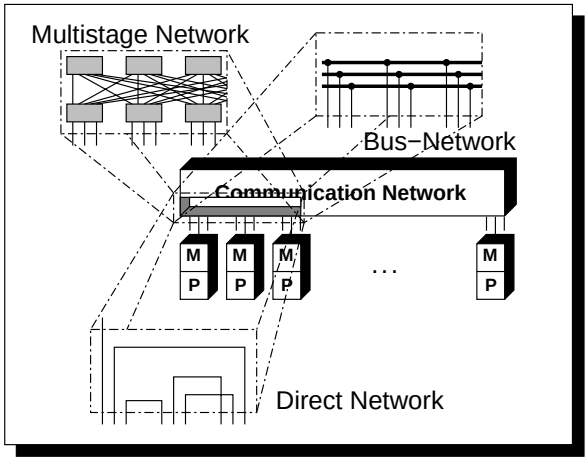

Figure 1: Variants of the DMM.

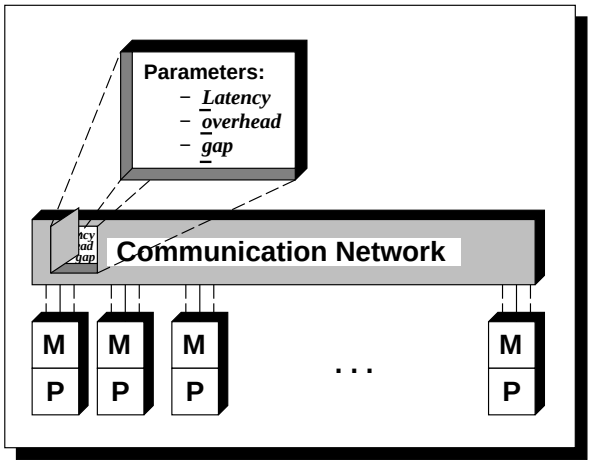

Figure 2: The LogP-Model.

2 Load Balancing: A Classification

2.1 Modeling Architectures and Applications

To be able to classify load balancing problems, one has to define models describing the application and the architecture of the target machine. The exchange of data in parallel systems with distributed memory is usually performed by communication via a routing network. Therefore, for the rest of the paper we will consider only message-passing based systems. Such architectures consist of a number of processors with local memory and a communication network. Concepts like virtually shared memory or global address spaces which appear in some of the most recent commercial systems are usually implemented by message passing, too.

Figure 1 shows different variants of communication networks. They can, for example, be bus networks like in the Intel Paragon machines, multi-stage networks like in the Thinking Machines CM5, the Parsytec CC, the Meiko CS2, or the IBM SP, or direct networks like in the older T800-based transputer systems, the NCube machines, or the Cray T3D/E.

Several models describing parallel systems and parallel applications have been designed in the past [54]. The most popular certainly are the *LogP*- and the *BSP*-model [14, 57, 78]. The models differ in the way how processors and communication networks are described. Most of them abstract from the structure of the routing network. Unfortunately, the communication structure of the application and the communication capabilities of the network are important factors for the design of load balancing algorithms and can not be neglected.

The LogP-model [14] describes a parallel system with help of four parameters (Fig. 2). P is the number of processors, the other three (*latency, overhead* and *gap*) describe the communication capabilities: L is the communication latency, i.e., the time the network needs to transmit a message of unit size. The network needs additional o(verhead) cycles

to set up a communication and $g(\text{ap})$ cycles have to pass between two consecutive messages in order to avoid an overloading of the network (because in this case the latency would increase and L would not be a valid parameter any more [58]).

The BSP-model [78] is rather a programming model and requires the parallel system to provide asynchronous communications and to be able to operate in lock-steps. The model distinguishes between calculation and communication phases. Communications can be set up during a calculation phase. A following barrier-synchronization between all processors ensures that all messages are delivered afterwards. Many applications can be modeled in this lock-step fashion and for analyses, the number of phases are counted. Extensions of this model include more detailed parameters of the communication behavior of the parallel system [3].

The *Distributed Memory Model (DMM)* [47, 57] is suited best to describe the topological structure of a parallel system (cf. Fig. 1). It describes the machine by a graph where nodes stand for processors and edges denote possible connections.

In the following, we view a parallel system with P processors as a graph $H = (U, F)$ with nodes $U = \{0, \ldots, P-1\}$ and edges $F \subseteq U \times U$. We consider only homogeneous systems, thus we omit node and edge weights which would be needed to model heterogeneous machines.

A parallel application is modeled as graph $G = (V, E, \rho, \sigma)$ with nodes $V = \{0, \ldots, N-1\}$, edges $E \subseteq V \times V$, node weights $\rho : V \to \mathbb{R}$ and edge weights $\sigma : E \to \mathbb{R}$. The meaning of nodes and edges can differ from application to application. For example, nodes can describe processes or data items, edges can stand for data dependencies or communication demands.

2.2 Classification

Using the modeling of Section 2.1, load balancing can be viewed as a graph embedding problem. The task is to find a mapping $\pi : G \to H$ of the application graph to the processor graph minimizing certain cost criteria. Depending on the application, the graph G can be *static* or *dynamic*, i.e., the computational load of the application nodes may or may not change during run-time. The processor graph is usually considered to be static.

Static applications have to be mapped only once onto the processor network and do not change during run-time. Thus, the load balancing problem can be reduced to a classical mapping problem where one graph has to be embedded into another. Most applications from the area of scientific computing behave static in this context. Examples are all kinds of non-adaptive FEM-simulations [17]. Section 3 considers the case of static load balancing and presents some of the most efficient solution methods.

Dynamic load balancing problems occur if the application graph changes during run-time. The graph may grow or shrink, i.e. nodes and edges might be inserted or deleted, or the node and edge weights may vary. Load balancing problems can be classified according to the weight functions of the application graph. Especially the *granularity* of an application

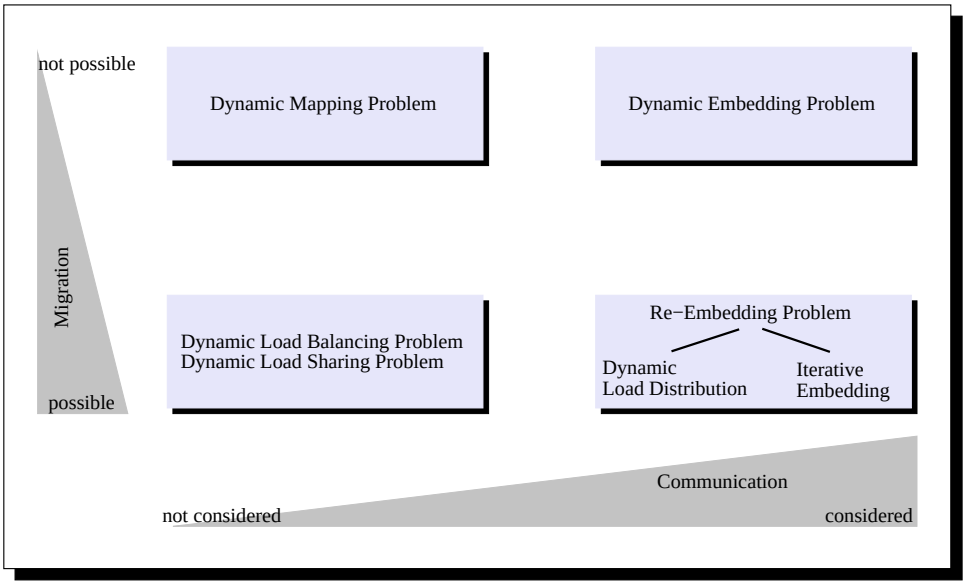

Figure 3: Classification of dynamic load balancing problems.

is of importance for the choice of the right balancing algorithm. Fine-granular applications consist of a large number of light-weighted nodes representing small processes or data items. Coarse-granular tasks are those with a small number of heavy processes.

The distinction often determines whether an application node can be migrated or not. Heavy processes are usually difficult to move from one processor to another during run-time, while data items can be transferred without problems. Accordingly, we classify dynamic load balancing problems depending on the communication structure of the application and on the possibility to migrate processes. Figure 3 shows the subdivision of balancing problems into four classes. In the following, we briefly describe applications, balancing algorithms, and theoretical results for each of the left- and top-most three classes. Dynamic applications from scientific computing usually cause load balancing problems belonging to the class of *Re-Embedding*. This class will be considered in Section 4.

2.3 The Dynamic Mapping Problem

The *Dynamic Mapping Problem* usually appears in the context of client-server applications. During run-time, processes are generated dynamically and have to be placed onto processors. There are almost no dependencies between them (thus, only little communication), but they also can not be migrated. Once placed, a process has to stay on "its" processor until termination. Load balancing algorithms have to take into account the trade-off between increasing overhead (e.g. to gather information about the system state) and increasing balancing quality [16].

Many theoretical results could be found on this kind of problems in the recent years. The mapping problem can be modeled as a *"balls-in-bins" game* where a number of balls

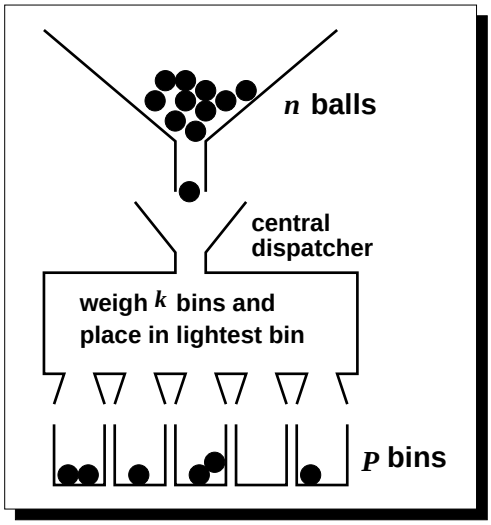

Figure 4: Dispatcher model for the dynamic mapping problem.

(processes) have to be placed into a number of bins (processors) (Fig. 4). For theory, it is (up to now) assumed that the balls are given in advance, placed by a centralized or distributed algorithm (the dispatcher), and that no load is consumed. It is known that if n balls are placed into n bins at random, then the fullest bin receives $O(\frac{\log n}{\log \log n})$ balls (with high probability (whp.) for $n \to \infty$ [2]). Azar et. al. could show that if for each placement, $k \geq 2$ bins are asked for their load and the ball is placed into the lightest, then, whp. and for $n \to \infty$, the fullest bin contains only $\frac{\log \log n}{\log k} + O(1)$ balls (which is an exponential increase in balancing quality) [2]. Extensions of the results to a parallel placement of balls are known [1], as well as optimal parallel algorithms for the placements of weighted balls [7]. Decker et. al. introduce additional costs for asking processors for their load and give best values for k depending on the number n of jobs and the communication capabilities of the parallel system [16].

2.4 The Dynamic Embedding Problem

If there are strong communication dependencies between the processes and they additionally can not be moved, the placement problem becomes quite difficult. This so called *Dynamic Embedding* problem occurs in dynamic tree-search applications like configuration systems, game tree search [59], or backtrack search [67]. The currently available balancing strategies place dynamically generated jobs onto processors in the neighborhood of their origin. Analytical results concerning load and dilation for balancing dynamic tree-like applications on "hypercubic" networks are known [36, 48, 61, 67]. First heuristics for grid-like applications appear, too [6] (which is then, of course, of relevance for certain scientific computing applications). But it turns out to be much more difficult to obtain analytical results if the application graph G is not as sparse as a tree.

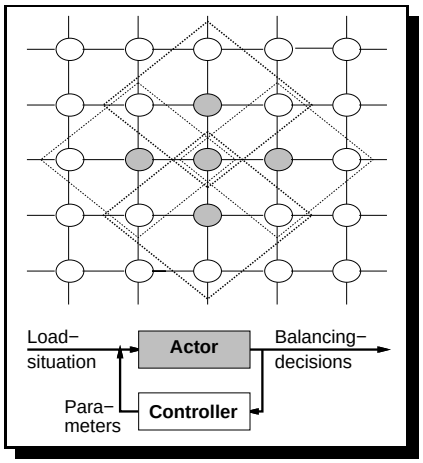

Figure 5: Balancing in overlapping islands and parameter control.

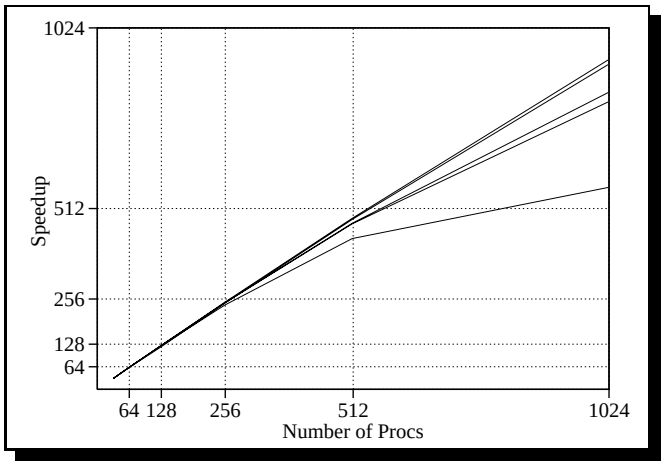

Figure 6: Speedup of B&B for the Vertex Cover problem on ≤ 1024 processors (using the LB-algorithm of Figure 5).

2.5 The Dynamic Load Balancing Problem

The *Dynamic Load Balancing Problem* is *the* classical load balancing problem. For applications where migratable jobs without communication dependencies are generated dynamically, a large number of efficient load balancing algorithms are known [9, 50, 51, 53, 77, 89]. A survey and classification concerning the type of information used to decide which load to move and the set of processors allowed to migrate to is given in [53].

Lin and Keller designed a gradient model for this balancing problem [50]. They assign a status of high, medium, or low to processors depending on their load. The algorithm then tries to push load from high to low. Ramme extended this model by additionally pulling load into the direction of "low" processors [53]. In Lüling's algorithm, processors balance their load with a fixed set of neighbors if the load difference between them increases above a certain threshold [51]. By adjusting the threshold to the migration frequency, thrashing effects could be avoided (cf. Fig. 5). For parallel branch & bound applications, the algorithms showed speedups of over 900 on 1024 processors (cf. Fig. 6 and [51]).

First theoretical results for a fully dynamic load balancing algorithm were given in [71]. The authors showed that if processor i initiates a balancing action with a randomly chosen other processor with probability $\frac{c}{load_i}$, then the expected load of i is at most c times the average load (plus a constant).

Lüling used a model where a processor balances with d (randomly chosen) others, if its load changes by a certain factor of f [52]. He could show that the expected amounts of load of arbitrary pairs of processors differ by not more than a factor which depends on f and d only.

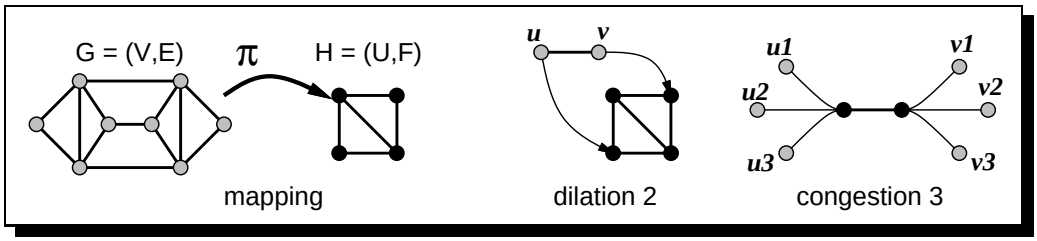

Figure 7: Mapping, dilation, and congestion.

3 The Static Problem

3.1 Mapping

This section deals with the static load balancing problem, i.e. neither the structure nor the weights of the application graph G change during run-time. We assume that G is completely known prior to the start of the application. Examples of this kind of applications occur in different fields, e.g. none-adaptive methods for numerical simulations.

The static load balancing problem is to calculate a *good* mapping of the application graph $G = (V, E)$ onto the processor graph $H = (U, F)$. In a formal description, a mapping is a function $\pi : V \to U$ which assigns each node of G to a node of H. Let $\delta : E \to \mathcal{P}(F)$ be a *routing function* which assigns each edge $e = \{u, v\} \in E$ to a path $w_e = (e_1 = \{\pi(u), q_1\}, e_2 = \{q_1, q_2\}, \ldots, e_i = \{q_{i-1}, \pi(v)\})$ with $|w_e| = i$.

Cost criteria determining the quality of a mapping are its *load*, *dilation*, and *congestion* (cf. Fig. 7 and [47, 57, 59, 60, 70]). The *load* of a mapping π is the maximum number of nodes from G assigned to any single node of H. Formally:

$$load(\pi) = \max_{p \in U} \sum_{v \in V, \pi(v) = p} \rho(v) \ . \tag{1}$$

The *dilation* is the maximum distance of any route of a single edge from G in H:

$$dil(\pi) = \max_{e = \{u,v\} \in E} |w_e| \ . \tag{2}$$

The *congestion* is the maximum number of edges from G that have to be routed via any single edge in H:

$$cong(\pi) = \max_{\bar{e} = \{p,q\} \in F} \sum_{\substack{e = \{u,v\} \in E \\ \bar{e} \in w_e}} \sigma(e) \ . \tag{3}$$

The load determines the balancing quality of the mapping. It should be kept as low as possible to avoid idle times of the processors (nodes of H). For the edge-measures dilation and congestion, we assume that every edge of G is used for the same amount of messages. Then, the dilation of an edge of G determines the slow-down of a communication on this edge due to routing latency in H. Clearly, the maximum dilation should be kept small, as in a synchronized model it is responsible for the length of a communication phase (if

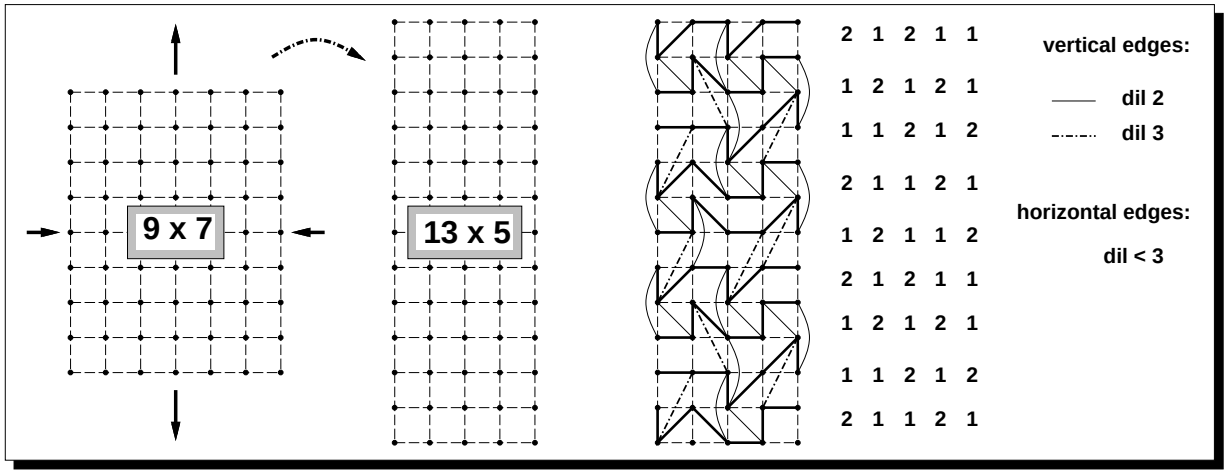

Figure 8: Mapping of a 9×7 grid onto a 13×5 grid.

congestion is neglected). Congestion on edges of H also may slow down the communication. If we assume the same data-throughput on each edge of H, and if a communication in G sends just this amount of data in each step, then a congestion of c on an edge of H will mean that completing all the c communications of G routed over this single edge takes at least c times the time it would need if there was no congestion.

Altogether, the goal is to find a mapping function π which minimizes all three measures - load, dilation, and congestion.

Some optimal mapping functions concerning load and dilation are known for well defined pairs of graphs like *Grids, Trees, Hypercubes*, etc. [47, 60]. As a small example we show in Figure 8 (following [69]) how to find a one-to-one embedding of $a \times b$ grids onto $x \times y$ grids with $x = \lceil \frac{ab}{y} \rceil$. The idea is to "push and pull" the $a \times b$ grid into the right shape. More formally, the rows of the $a \times b$ grid are folded into the $x \times y$ grid. The prescription of how to fold is given by an *embedding matrix* (right in Fig. 8) containing y columns and a rows. Row i describes how to fold row i of the $a \times b$ grid. The contents of row i in column j of the matrix give the number of nodes from row i of the $a \times b$ grid to place in column j of the $x \times y$ grid. A number $k > 1$ in an odd column means to place the k nodes from "top-to-bottom"; in an even column the placement is "bottom-to-top". The first row of the embedding matrix is given by

$$\left(\left\lceil \frac{b}{y} \right\rceil, \left\lceil \frac{2b}{y} \right\rceil - \left\lceil \frac{b}{y} \right\rceil, ..., \left\lceil \frac{(j+1)b}{y} \right\rceil - \left\lceil \frac{jb}{y} \right\rceil, ..., \left\lceil \frac{yb}{y} \right\rceil - \left\lceil \frac{(y-1)b}{y} \right\rceil \right)$$

A row i can be obtained by a cyclic right shift of row $i-1$. It can be shown that the sum of each row equals b and the sum of each column of the matrix equals $\lceil \frac{ab}{y} \rceil$ or $\lfloor \frac{ab}{y} \rfloor$. The maximum dilation of this embedding is $\lceil \frac{b}{y} \rceil + 1$ [69].

This is only a small example for embedding grids into grids of a certain type. For special cases of ratios between (a, b) and (x, y), other methods are known which give similar or, sometimes, better results [69]. And, of course, many other results for mapping different

well defined graphs onto each other are known. For a survey, the book of Leighton [47] and the paper by Monien and Sudborough [60] may be used. Several of the methods are calculable in parallel and are implemented in libraries like for example described in [70].

If G or H do not belong to these well defined classes of graphs, heuristic methods have to be used to find a good mapping function. The general mapping problem is *NP*-complete [19], even if only load and dilation are considered as cost function. This means that all currently known solution algorithms need a run-time which grows exponentially with the size of the guest graph G, and, additionally, it is very unlikely that fast algorithms exist [29, 49]. Thus, optimal solutions for the general mapping problem can only be determined for very small graphs.

Meta-heuristics like *Simulated Annealing*, *Genetic Algorithms* or *Tabu Search* can be applied to the mapping problem [19, 20]. Such heuristics try to minimize a cost function consisting of a combination of load, dilation, and congestion. They are usually quite slow and their efficiency in terms of run-time and solution quality heavily depends on the way how the problem is coded and how a couple of parameters are adjusted [49].

The self-organizing feature mapping of Kohonen has been used for the graph-mapping problem by Heiß and Dormanns [32]. This iterative method is quite well suited for the mapping problem, although it is not able to guarantee balanced load.

Summarizing, we see that the task of mapping graphs onto graphs is solved for many special cases, but the general problem is (currently) impossible to solve to optimality. Even heuristics approximating the optimal solution are not efficient. Fortunately, recent developments in interconnection technology of parallel systems allow a careful relaxation from the strict definition of the mapping problem. Current parallel machines (often) contain communication networks consisting of routing switches which implement very efficient protocols like *Worm-hole-* or *Cut-Through Routing* [26, 68]. With these networks the definition of dilation is not longer valid and thus it is possible to relax from the topological structure of the host graph H [58]. It can in principle be viewed as a *clique*, a completely connected graph, leaving congestion as the only determining communication cost value of a mapping. Unfortunately, this is not completely true, as a processor is not able to communicate with an arbitrary number of others in one step. Thus, the number of messages, which a processor has to send, has to be considered, too. The next section shows the implications on the mapping problem.

3.2 Mapping in the LogP-Model

Multiprocessor systems with indirect routing networks like the Cray T3D/E, IBM SP, Parsytec GC/CC, and even the HP/Convex SPP and the SGI Origin can be modeled as clique graphs, i.e. (in the notation of Sec. 2.1) for all $p, q \in U$ it is $\bar{e} = \{p, q\} \in F$. Due to the very efficient networks and routing algorithms, there is no real "neighboring"-relation between processors any more. Communication between any two pairs is possible at nearly the same speed. Thus, the topological structure expressed by the DMM is no

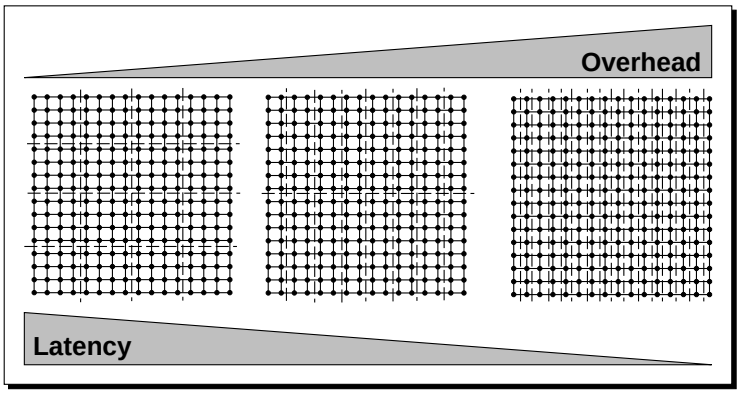

Figure 9: Overhead-Latency trade-off in the LogP-model leads to different mappings.

longer relevant. Instead, the number and size of messages a processor has to send can be considered. The size of messages is a measure to express congestion. As all messages on one edge of H go to the same target (there is no routing via nodes of H), they can be combined.

The LogP-model with its parameters L, o, and g is suited to analyze algorithms under such conditions. Sending a message of size m takes time $m \cdot L + o$ (if we assume that a processor sends a message every g steps at most, cf. Sec. 2.1). If a mapping of nodes from G to nodes form H has to be found, the trade-off between o and L has to be considered. Optimizing for o would mean to send as few messages as possible, considering L means to send as small messages as possible. Figure 9 shows this trade-off using the mapping of a 16×16 grid onto 16 processors. If the latency L is large, the size of messages should be kept small. Thus, the congestion of the mapping has to be considered (which is 4 in the leftmost mapping). If, on the other hand, the overhead o is large, a processor should set up as few communications as possible. Thus, the maximum degree of the remaining *quotient graph* of the mapping (which would be a 4×4 grid in the left mapping of Figure 9 and a 1×16 grid in the right) has to be small.

The mapping problem restricted to the LogP-model turns out to be some kind of a partitioning problem: The guest graph G has to be split into a number of parts such that the degree of the resulting quotient graph and the maximum number of edges of G running between any pair of partitions are small. We may further relax the problem to the classical partitioning problem [49] by ignoring the parameters L and o of the LogP-model and by just requiring to have as many edges of G internal as possible. Using the notation of Sec. 2.1, the partitioning problem can be formulated as follows: The graph G has to be partitioned into $|U|$ equally sized *parts* $V_0, V_1, ...V_{|U|-1}$ such that only very few edges are external. Instead of the congestion (3), the *cut* of the partition has to be considered:

$$cut(\pi) \;=\; \sum_{\bar{e}=\{p,q\}\in F} \; \sum_{\substack{e=\{u,v\}\in E \\ \bar{e}\in w_e}} \sigma(e) \quad = \sum_{\substack{e=\{u,v\}\in E \\ \pi(u)\neq\pi(v)}} \sigma(e) \tag{4}$$

The partitioning problem is *NP*-complete (even for the case of $|U| = 2$, [29]), but there

exist a large number of efficient heuristic methods which minimize the load as defined in (1) as well as the cut according to (4). The following section introduces the most relevant existing partitioning heuristics.

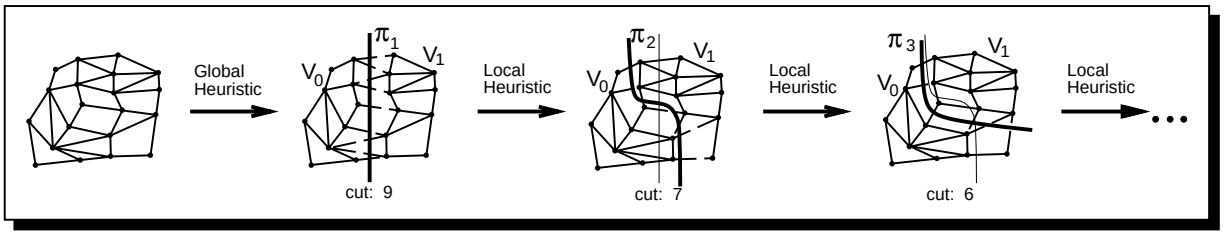

Figure 10: Global and local bisection heuristics.

3.3 Graph Partitioning

The graph partitioning problem as defined in (4) is to split a graph into as many parts as there are numbers of processors such that as few as possible edges are external. This so called *k-partitioning problem* is still hard to solve directly but can be done by recursively *bisecting* the graph into two pieces. For the bisection problem, many efficient solution heuristics exist which do not solve it to optimality (as it is still *NP*-hard), but approximate the best value. The methods differ in their solution quality in terms of numbers of external edges (the *cut*, (4)), in the time they need, and, sometimes, in the additional information they require.

Bisection heuristics are usually divided into *global* and *local* methods (cf. Fig. 10 and [22, 33]). Global methods are sometimes called *construction heuristics*, because they take the graph description as input and generate a partition. Local methods are called *improvement heuristics*. They take the graph and a partition as input and try to improve the cut by local re-arrangements of nodes. We will describe existing local and global methods in the next two sections.

3.3.1 Global Methods

Global bisection methods have to construct a valid bisection of a graph, i.e. they have to generate a partition of the nodes into two subsets of equal size. The currently available methods use different amounts of information on the graph. They range from very simple methods which just use node numberings to very sophisticated heuristics using information on the topological structure of the graph. Accordingly, their time and space requirements differ as well as the quality of solutions they generate.

Very simple methods just use the numbering of nodes of the graph to split it into two parts. They generate partitions of nodes with odd (even) indices, or with indices smaller (larger) than half the number of nodes. Such methods are very fast, but the quality of

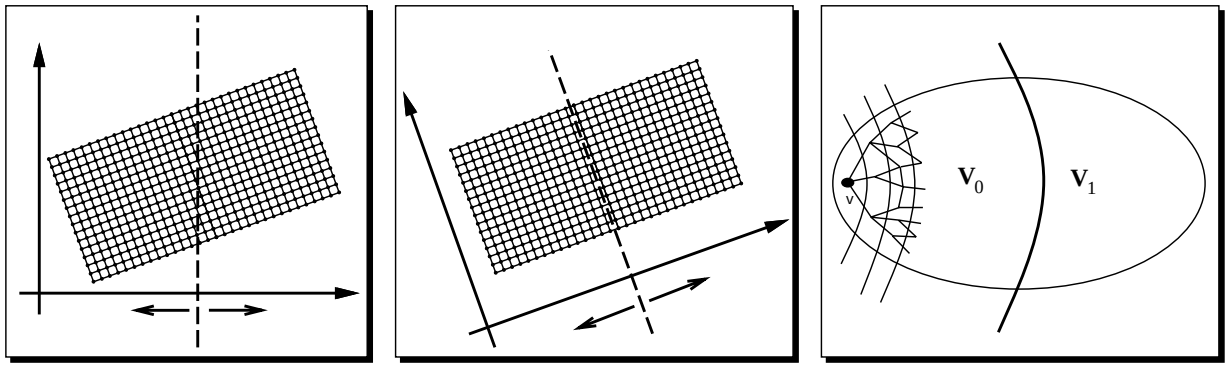

Figure 11: Co-ordinate, Inertial, and Greedy Bisection.

solutions they create depends heavily on the numbering of nodes and thus on the way the graph is generated [66]. Usually, these methods are pre-steps to sophisticated local improvements heuristics.

If the graph is a mesh, i.e. if it is embedded into the Euclidean space and geometric co-ordinates are attached to its nodes, then this extra information can be used to generate slightly better partitions. Simple co-ordinate bisection methods sort the nodes according to their x-, y-, and z-co-ordinates and then split the graph in the x-, y-, or z-direction depending on its elongation in either of the three directions. Figure 11 shows a simplified example for the 2D case. The method can be applied recursively in a balanced or unbalanced way [41, 42] to cut into more than two parts, and it is also possible to directly split into a number of partitions [21].

A slightly more elaborated method first rotates the co-ordinate system according to the main principle axis of the graph before using co-ordinate bisection (Fig. 11). The *Inertial* bisection heuristic calculates the center of gravity of the graph as the (weighted) mean of all the node co-ordinates. Using the Euclidean (or L_2) norm, it then measures the distance of all nodes from the center point. The accumulated distances in x-, y-, and z-directions, and all possible pairwise combinations of x, y, and z are put together to form a 3×3 matrix. The main eigenvector of this matrix determines the principle axis of the graph. It is then partitioned orthogonal to this axis [33, 84].

A different class of methods does not use geometric information of the graph but the connectivity instead (Fig. 11). The so-called *greedy* methods build up a bisection starting from one node (as V_0) by repeatedly adding the node which increases the cut size least [65]. Farhat uses an extension of this algorithm to partition into k pieces directly [24, 25]. Other versions determine several of the starting nodes first and build up each part in parallel [73].

A widely used global partitioning heuristic is the *Spectral Method* which is based on algebraic graph theory [4, 64, 84]. To get an idea of how this method works, we have to re-write the bisection problem in a matrix notation. Let A be the *adjacency matrix* of the graph G, i.e. $A = (a_{uv})$ with $a_{uv} = 1 \Leftrightarrow (u, v) \in E$ and $a_{uv} = 0$ otherwise. Let D be a diagonal matrix with $d_{uu} = deg_u$, i.e. D contains G's node degrees. The matrix $L = D - A$

is called the *laplacian matrix* of G. A bisection π can be written as vector x with $x_v = \pm 1$ where $V_0 = \{v; x_v = -1\}$. If x is a bisection (w.l.o.g. assume $|V|$ even), then $\sum x_v = 0$. The cut size from (4) can be re-written to

$$cut(x) = \frac{1}{4} \sum_{\{u,v\} \in E} (x_u - x_v)^2 \ . \tag{5}$$

It is an easy exercise to see that (5) is equivalent to $cut(x) = \frac{1}{4} x^t L x$. The bisection problem is then to find a vector x minimizing $cut(x) = \frac{1}{4} x^t L x$ such that $\sum x_v = 0$ and $x_v = \pm 1$. This discrete problem is of course difficult to solve (as bisection is *NP*-hard), but we can do a relaxation of the condition $x_v = \pm 1$ to $x_v \in I\!R$ with $x^t x = n \ (= |V|)$. Let $\tilde{x}$ be the relaxed bisection vector. $\tilde{x}$ is an approximation to the real bisection vector x. The bisection can be obtained from $\tilde{x}$ in different ways. It is for example possible to sort the entries in $\tilde{x}$ and to take the first half as V_0 and the second half as V_1. The problem which remains to be solved is to find a good $\tilde{x}$. The Spectral method reduces this task to an Eigenvalue problem.

Let $\lambda_1 \leq \lambda_2 \leq \ldots \leq \lambda_n$ be the eigenvalues of L and $\underline{u}_1, \ldots, \underline{u}_n$ be the corresponding normalized eigenvectors (as $\underline{u}_2$ is of special interest here, we will use $\underline{y} := \underline{u}_2$ in the following). It is known that $\lambda_1 = 0$, $\lambda_2 > \lambda_1$ (if G is connected), and $\lambda_n \leq 2 \cdot deg_{\max}$. The eigenvectors of L form a basis of the $I\!R^n$, so we can express $\tilde{x}$ as $\sum \alpha_i \underline{u}_i$ with some $\alpha_i \in I\!R, \sum \alpha_i^2 = n$. Some easy calculations then show that

$$\frac{1}{4} \tilde{x}^t L \tilde{x} = \frac{1}{4}(\alpha_2^2 \lambda_2 + \ldots + \alpha_n^2 \lambda_n) \geq \frac{1}{4}(\alpha_2^2 + \ldots + \alpha_n^2)\lambda_2 = \frac{n\lambda_2}{4} \tag{6}$$

which is a lower bound on the minimal number of edges which have to be cut by any bisection. Using $\tilde{x} = \sqrt{n} \cdot \underline{y}$ gives equality in (6) which shows that splitting according to $\underline{y}$ is the best possible solution of the approximation problem. A projection to the discrete case can be done as described above.

The second smallest eigenvector $\underline{y}$ expresses the *algebraic connectivity* of the graph and is sometimes called *Fiedler*-vector [27, 64]. Fiedler showed in '73 that if G is connected and r is any real number, then the sub-graph $V_0 = \{v; \underline{y}_v \geq -r\}$ is connected [27]. This gives some intuitive hint on why the approximation of x by $\underline{y}$ might not be too bad and, additionally, enables a recursive application of the method to split into more than two parts.

The Spectral method as originally described in [64] determines $\underline{y}$ to L and splits the graph according to the entries in the vector. The results are usually fairly good, especially globally seen. Due to the projection to the discrete case, local improvements are often possible.

Determining the eigenvectors of L is an expensive task which limits the usability of the method. Therefore, in implementations of libraries like *Chaco* [33], efficient incomplete orthogonalizations are used or multi-level solvers as described in [4]. Additionally, certain attempts are made to use more eigenvectors to directly split into four or eight parts [35].

3.3.2 Local Methods

Although some of the global partitioning methods usually produce good cuts, there is often a large potential for further improvements by local rearrangements. Local heuristics determine equally sized sets of nodes which can be exchanged between parts such that the size of the cut decreases (cf. Fig. 10). The methods differ in the way how these sets are found. Simplest local search just tries to exchange pairs of nodes until no further improvement is possible [49]. This method is quite fast but the possible improvements are limited due to the existence of *local minima*, i.e. partitions where all possible swaps of nodes would increase the cut. The example in Figure 12 shows such a situation.

The *Kernighan-Lin* heuristic [45] (or, shortly, the KL-algorithm) uses the same basic operation - swap of node pairs - but it allows intermediate increases of the cut size, thus avoiding certain local minima. Figure 12 shows an example of how the KL-algorithm works. Let C be the cut size and $b(u, v)$ the *benefit* of exchanging u and v, i.e. the decrease in cut size if u and v are swapped. Note that $b(u, v)$ does not have to be positive. The KL-algorithm chooses in each swap the pair of nodes with the largest benefit (which might be < 0), exchanges them logically and locks them. These swaps are repeated $\frac{n}{2}$-times until all nodes are locked. Afterwards, the algorithm starts from the beginning searching for the sequence of logical swaps with the largest accumulated benefit, i.e. the lowest cut size. This sequence of swaps is then performed physically. This "super-step" of searching for a good sequence of swaps is repeated until no further improvement is achieved.

The original KL-algorithm as described above (and in [45]) is quite slow. The time needed for one "super-step" on a graph with n nodes is $O(n^3)$ ($\frac{n}{2}$ times searching for the pair with maximum benefit). Fiduccia and Mattheyses in '82 described an improvement to $O(n^2)$ per "super-step" ($O(n)$ on sparse graphs) by not searching for pairs of nodes with maximum benefit but just taking single nodes on each side [28]. For real implementations a lot of extra "tricks" like randomization and early termination of "super-steps" improve the run-time and solution quality substantially [33, 34, 65].

An alternative local improvement method is the *Helpful-Sets* heuristic [22, 56]. It proceeds in a two step manner. The first step tries to find a set of nodes in one part and moves it to the other in order to decrease the size of the cut while neglecting the load. The second step tries to re-balance the load while keeping the size of the cut below the initial value. Figure 14 shows the main idea of the algorithm. The basic operation is no longer a swap of nodes but a move of certain subsets. The subsets have to be *helpful* [38]. We will describe in the following what this means.

For each node v of a graph G with partition π define $ext(v) = |\{u; \{u, v\} \in E, \pi(u) \neq \pi(v)\}|$ as the number of external edges incident to v. Consequently, $int(v)$ is its number of internal edges and it holds $deg(v) = int(v) + ext(v)$. Then, $h(v) = ext(v) - int(v)$ is called the *helpfulness* of node v and v is called $h(v)$-helpful. The helpfulness of a node is equal to the change in cut size if this node is moved to the other subset. The nodes in Figure 13 are marked with their helpfulness.

Furthermore, let $S \subset V_i, i \in \{0, 1\}$ be a subset of nodes from one side of the cut. We

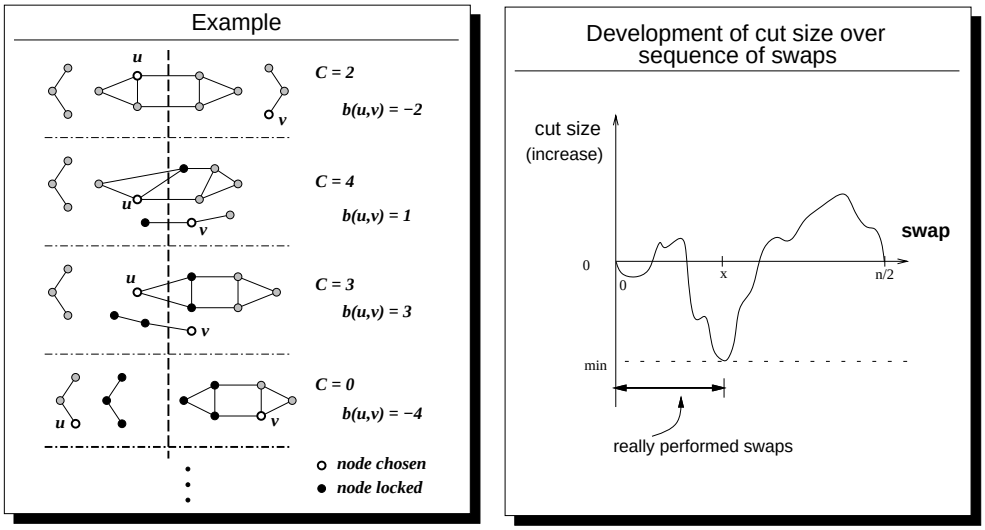

Figure 12: The KL-algorithm.

define $int(S) = \sum_{v \in S} |\{\{u, v\} \in E; u \in V_i \wedge u \notin S\}|$ to be the number of internal edges of S, i.e. edges which lead out of S but to nodes on the same side of the cut. Similarly, $ext(S) = \sum_{v \in S} |\{\{u, v\} \in E; u \notin V_i\}|$ is the number of external edges of S, i.e. the number of edges out of S crossing the cut. Then, $h(S) = ext(S) - int(S)$ is the helpfulness of set S and S is called $h(S)$-helpful. If S is moved completely from V_i to V_j, the cut size decreases by $h(S)$ edges. Figure 13 shows a number of 2-helpful sets. The examples indicate the strength of the method: nodes which never would be moved by their own might contribute to a helpful set and therefore can be moved together with others.

Another strength of the method comes from theory. Using the definition of helpfulness, it is possible to show an upper bound on the bisection width of $2k$-regular graphs (i.e. it is possible to show how many of the edges of a given graph have to be cut at most in order to construct a bisection) [56]. The proof of this bound consists of two steps. It shows that if there is a bisection and if its cut size is not too small, then there exists a small 4-helpful set in either V_0 or V_1. This 4-helpful set can be moved to the other side and decreases the cut size by 4. The second step shows that if the imbalance is not too large and if the cut size of such an unbalanced partition is not too small, then it is possible to balance the partition by moving enough nodes from the larger to the smaller part without increasing the cut size by more than 2 edges altogether. Both steps can be applied iteratively until the "too small" condition on the number of external edges is no longer fulfilled. The bound which can be proven by this method states that for any n-node $2k$-regular graph (a graph with n nodes where each node has degree $2k$), at most $\frac{k-1}{2}n + 1$ edges have to be cut by a bisection. The formal proof of this theorem is given in [56].

The HS-heuristic implements this proof technique [22, 56]. Figure 14 shows the principle. The heuristic searches for a small helpful set S on both sides of the cut. If such a set is found, it is moved to the other side. The algorithm then searches for a balancing

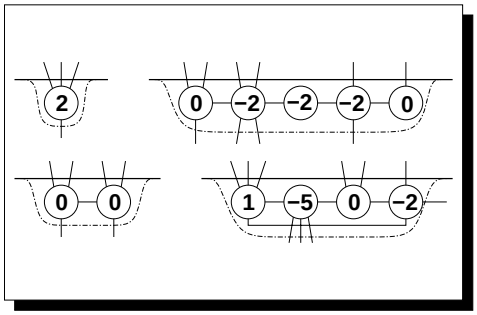

Figure 13: Some 2-helpful sets.

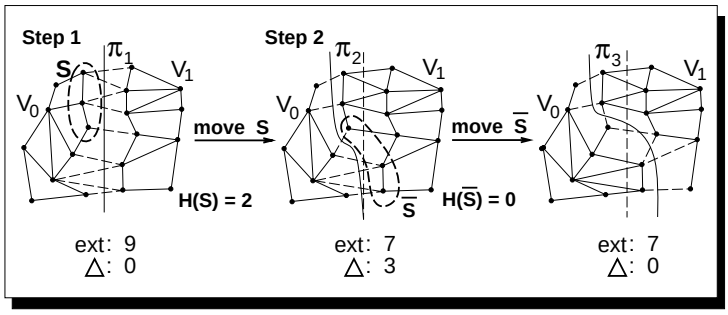

Figure 14: The HS-algorithm.

set $\bar{S}$ on the larger of the two sides. $\bar{S}$ has to have the same size as S but has to be at least $(-h(S)+1)$-helpful. Both steps together construct a new bisection whose cut size has been decreased by at least one edge.

The HS-heuristic is included in the PARTY partitioning library [65, 66]. The time needed for one iteration is $O(n)$ on sparse graphs. Several tricks of the implementation, such as searching for c-helpful sets with a number $c \gg 4$ increase its speed of convergence [22]. It behaves comparable to the fastest implementations of the KL-algorithm in both time and quality of solutions. On some examples, the KL is slightly better or faster, on others HS is preferable [56, 66].

3.3.3 Multilevel Hybrid Methods

If very large graphs have to be partitioned, coarsening techniques can reduce the time-requirements [4, 33, 43, 82]. The idea is to shrink a large graph to a smaller one with similar characteristics, partition it efficiently and extrapolate the partition to the original graph.

Figure 15 shows an example. There are many different possibilities for coarsening [43]. The example in the figure uses what is called *heavy edge matching*. It determines an independent set of edges, i.e. edges which are not "neighbored" and contracts them to a single node. Some of the remaining edges then fall onto each other. Edge weights can be used to express this multiplicity. While choosing edges for a matching, preference is given to heavy weighted edges.

Other possibilities are coarsenings by graph partitioning. Fast and simple partitioning heuristics can be used to split the graph into many small parts. Each part then forms a node of the next coarsening level.

The smallest graph can be partitioned using any of the presented global methods. Usually, Spectral Partitioning is used as it comprises the most global information of the graph [33, 43]. The coarsening steps preserve the main topological structure of the graph such that a global partition of the coarsest one is a good approximation to a global partition of the finest.

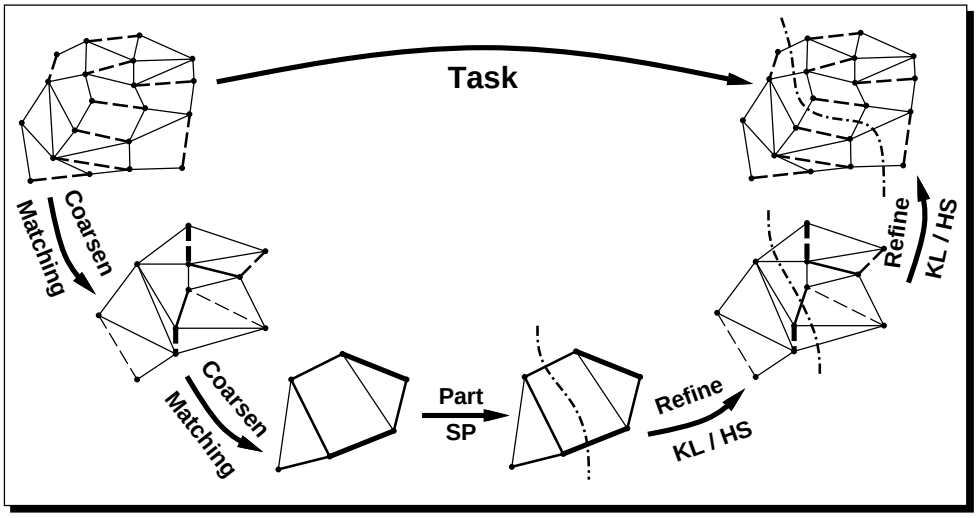

Figure 15: The multilevel algorithm.

The partitioning of the coarsest graph has to be able to deal with weighted nodes in order to ensure load balance. The node weights are assigned during the coarsening process and express the number of nodes of the finest graph represented by each node of the coarsest. If the matching technique is used and if the matchings are not perfect in certain steps (i.e. if not every node belongs to a matching edge), then the node weights differ from each other.

The refinement steps invert the coarsening operations. Nodes of coarser levels are expanded to edges or sub-graphs. If the coarse-graph partitioning did not consider the load because it was not able to deal with weighted nodes, the refinement steps have to move nodes across boundaries to balance the partition [44]. Additionally, a local heuristic like KL can be (and usually is) used to "smooth" the partition boundaries in each refinement step [34, 43].

Multilevel methods turn out to be the most powerful of the existing partitioning heuristics. Their time requirements are almost independent of the graph size. As they collapse the graph to a constant number of nodes and as the graph size (nearly) halves from level to level, their run-time is dominated by the time of the first coarsening step. On sparse graphs, this usually is $O(n)$.

Additionally, multilevel algorithms naturally include what is also advisable with all the other methods: A combination of global and local heuristics. In comparisons they usually produce solutions which are among the best of all methods. Section 3.3.4 shows some results of different methods implemented in different libraries.

3.3.4 Comparisons and Results

Although there are some theoretical results on the performance of specific partitioning heuristics, they are still very weak due to the broad range of possible application examples.

Usually, each application constructs examples with some specific characteristics and it is left to show that theoretical observations also hold true for implementations of the heuristics. Some of the heuristics are already very complex and are getting even more complex when implemented with additional generalizations. Therefore, there are many implementations details which influence the behavior of the heuristics.

Several software libraries have been developed to serve solutions to the graph partitioning problem. Examples are *Chaco* [33], *Jostle* [83], *Metis* [43], *Scotch* [63], and *Party* [65]. Their goal is to provide efficient implementations and to offer a flexible and universal graph partitioning interface to applications. Partitioning libraries contain several different heuristics which can further be tuned by adjusting specific parameters. Their performance steadily increased in the recent years, leading to the ability of constructing partitions with low cut sizes of very large graphs with more than a million nodes in the measurement of seconds. To reflect specific application constraints more precisely, they often include several extensions and generalizations like, for example, handling of node- and edge-weighted graphs, optimizing for certain processor topologies or considering different cost functions.

It is difficult to perform an objective comparison of the implemented heuristics, because there are many test cases one may consider and the libraries are designed with different applications in mind. We choose bisection as the atomic step for comparing partitioning heuristics. We consider different graphs from different applications and sources as benchmark suite. And we compare only those libraries which aim at being partitioning tools rather than mapping libraries. We take some of the key heuristics from the different tools and compare cut sizes and run-times on our set of benchmark graphs.

It is desirable for partitioning heuristics to have a stable and efficient performance when used for graphs of different character. Therefore, we choose graphs from flow simulation (*airfoil1, wave*), structural mechanics (*crack*), aircraft crew scheduling (*lh*, with edge weights), sparse matrix factorization (*mat03HBF*) and theoretical computer science (*DEBR20*).

We consider the Multilevel (ML) method from the Chaco library, which uses the spectral method on the coarse graph and the Kernighan-Lin (KL) refinement on several intermediate levels, and the Inertial (IN) method with and without additional KL. The Metis library serves with the multilevel strategy *pmetis*. Additionally, the Party library offers the setting *all*, which performs 5 or 6 different node numbering, greedy and, if possible, coordinate methods and takes the best result. This setting can also be combined with the local refinement method Helpful-Sets (HS), which is the default setting of Party. Please refer to the User-Guides of the libraries for more details.

Table 1 shows some characteristics of the graphs and the partitioning results. For graph *lh*, the total number of edges and the total number of edge weights are shown. The Inertial method can only be used for graphs with geometric coordinates.

None of the settings outperforms all others. The Inertial method is very fast and the cut sizes can significantly be decreased by applying the KL refinement. The multilevel strategies *ML* and *pmetis* compute low cut sizes with low run times. They only lack for

Graph	$\|V\|$	$\|E\|$	Chaco ML	Chaco IN	Chaco IN+KL	Metis pmetis	Party all	Party all+HS
airfoil1	4253	12289	85 (0.08)	94 (0.00)	83 (0.02)	85 (0.04)	94 (0.04)	83 (0.15)
crack	10240	30380	211 (0.16)	377 (0.01)	218 (0.05)	196 (0.14)	243 (0.10)	208 (0.44)
wave	156317	1059331	9542 (3.64)	9834 (0.19)	9660 (1.61)	9801 (3.50)	10361 (2.84)	9614 (11.93)
lh	1443	20148 487380	36376 (0.33)	– (–)	– (–)	22579 (0.06)	13643 (0.06)	9897 (0.23)
mat03HBF	73752	1761718	9359 (1.80)	– (–)	– (–)	9555 (2.04)	8869 (3.45)	8869 (11.52)
DEBR20	1048576	2097149	100286 (48.99)	– (–)	– (–)	101674 (988.39)	172204 (16.63)	94272 (577.97)

Table 1: Test graphs and cut sizes for partitioning into two parts. Numbers in brackets indicate run times in seconds on a Sparc Ultra 2.

the graph *lh* with weighted edges. The settings of Party seem to be quite robust, due to the performance of several different simple methods. Although their run times are usually higher than for the other settings, the resulting cut sizes are among the lowest if used with the local refinement HS, which, again, significantly reduces the cut sizes.

Overall, many different methods and implementations exist and the user may choose a setting appropriate for his application. If the calculation has to be very fast, simple methods like Inertial already produce partitions with reasonable cut sizes. But even for very large graphs, multilevel strategies and local refinement methods are fast and result in very low cut sizes. The combination of several methods of different character increases the run time, but also results in low cut sizes and an overall robust behavior.

4 Dynamic Load Balancing

4.1 The Problem

A classification of dynamic load balancing problems was given in Section 2. Here we will consider the *Dynamic Re-Embedding Problem* in more detail. The application graph $G = (V, E, \rho, \sigma)$ of problems in this class is dynamic, i.e. nodes and edges are generated or deleted during run-time. We assume that the applications operate in *phases*. Changes of G do not occur at arbitrary, non-predictable times but in a more or less synchronized manner. For algorithms in scientific computing, this is a realistic assumption. The mesh is usually refined based on error estimates of the current solution [5, 11]. Therefore, the calculation of the solution has to be finished before the error estimation and refinement can be performed.

Figure 16 shows an example of a mesh adapted towards a corner singularity. Especially if the singularity is moving, i.e. if in-stationary processes are supposed to be simulated, the structure of the mesh changes rapidly throughout the whole calculation.

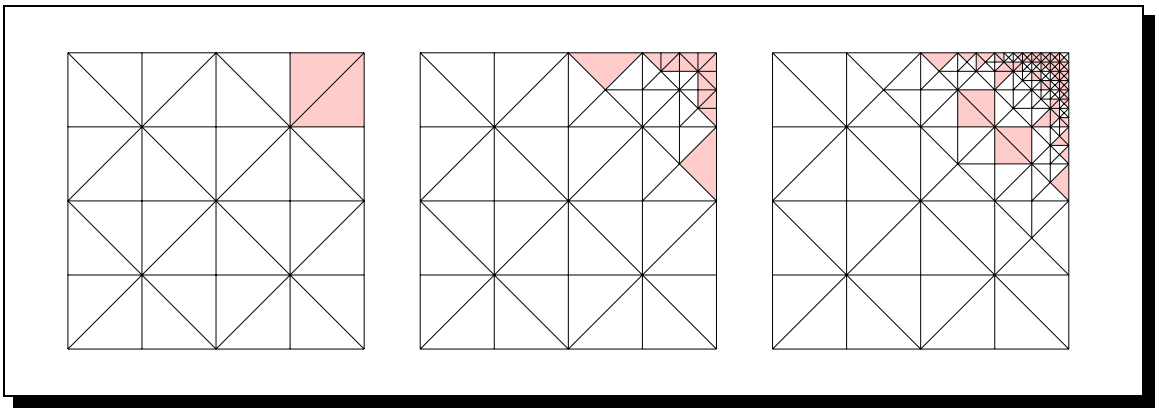

Figure 16: A sequence of mesh adaptations.

The synchronized occurrence of load imbalances results in a kind of synchronized load balancing problem at application determined times. The execution of the next calculation step of the application has to wait until the load balancing is completed. In this way, we know that during the process of balancing, no new load is generated and non is consumed.

One possible solution strategy for such types of synchronized balancing problems views them as a series of static ones. The load balancing is performed by only one processor using some of the partitioning heuristics we mentioned in Section 3 [75]. The scalability problems if only one processor is doing the work can be avoided by parallelizing the partitioning step [44, 62, 85]. Unfortunately, another difficulty imposes with this approach: The balancing does not take advantage of the knowledge included in the existing distribution of load over processors. Therefore, the new mapping often differs substantially from the current one and a large amount of load (data) has to be shifted between processors. Although there are heuristics minimizing the amount of load migration in this approach [62, 75], in many cases still a large percentage of data has to move. Walshaw et. al. made experiments comparing this kind of "re-partitioning" to dynamic load balancing algorithms which take knowledge about the existing data-distribution into account [83]. In some cases, they were able to reduce the amount of data-movement from over 80% to less than 10% for the same balancing problem.

In the following, we focus on "dynamic re-mapping" algorithms which start from a given distribution of data over processors and try to balance load while shifting as less data-items as possible. Due to the data-dependencies between load-items (elements of FE-meshes for example), it is not advisable to just place newly generated load onto arbitrary processors to ensure load balance (like it is in the dynamic mapping problem). This would impose heavy communication between nearly all pairs of processors after a short while. We could, of course, consider H as being a complete graph where such a kind of placement is possible (cf. Sec. 3.2). But as a migration of nodes from G should consider *data locality*, i.e. nodes should not be moved to processors where none of their neighbors are located, we have to reduce H to the *quotient graph* of G resulting from the mapping of G to H (i.e.,

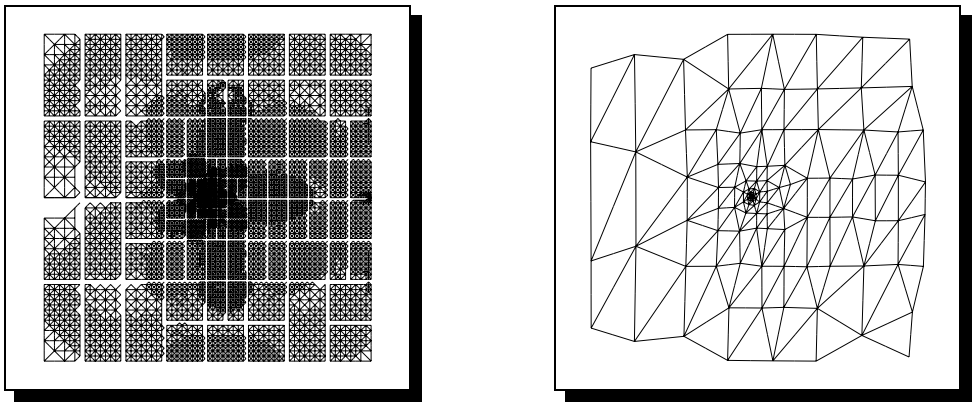

Figure 17: Mapping of graph *crack* to 128 processors (le) and resulting quotient graph (ri).

the structure of H now expresses the communication demands induced by the mapping, cf. Fig. 17). If the load balancing is performed on this quotient graph only, data locality can be guaranteed.

We can split the task of load balancing into two steps [85]. The first one calculates *how much* load has to be shifted between processors, the second step determines *which* load has to be moved [21].

4.2 "How Much"

4.2.1 Modeling as a Network Flow Problem

The first step of the load balancing, the "how much", can be viewed as a flow problem. For each edge of H one has to calculate how much load has to be transferred across it in order to achieve a globally balanced system. Figure 18 shows a example for this construction. Artificial source and sink nodes are added to the quotient graph. The source is connected

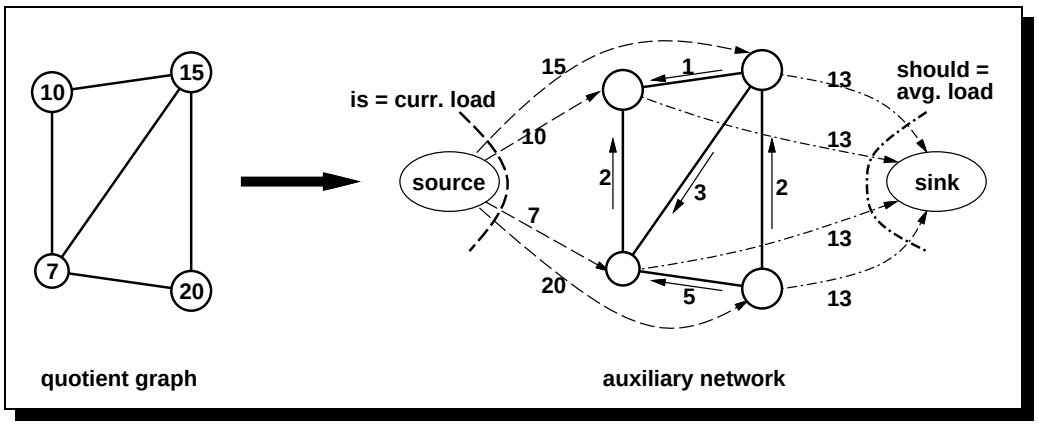

Figure 18: Modeling the Re-Embedding Problem as Network Flow Problem.

to each "normal" node by edges of capacity according to their load before balancing. The edges to the sink have capacity equal to the mean load value. Solving a network flow problem on this auxiliary network gives the amount of data to shift between any pair of neighbored processors in the quotient graph in order to achieve load balance.

Of course, we could use network flow algorithms to solve this problem. Unfortunately, they usually apply to much more complicated cases of flow problems and are quite slow. Furthermore, they are difficult to parallelize. For the case considered here an algorithm is needed which is easy to parallelize, scalable to large numbers of processors, and local in the sense that it does not need a global control and that processors operate with only a limited number of others.

4.2.2 Token Distribution

In theoretical computer science the synchronized setting of the Re-embedding problem is known as the *Token Distribution Problem*, where the task is to evenly distribute a number of independent tokens on a network of processors. For this problem, a large number of theoretical results as well as algorithms are known [55]. Counting the number of tokens sent by such an algorithm via each edge immediately gives a solution to the question of "how much". The theoretical analysis of the token distribution algorithms counts synchronized rounds of the network wherein only one token is allowed to traverse each edge per step. Thus the results only give upper bounds on the time needed to determine the balancing flow.

4.2.3 Random Matchings

Interestingly, relatively simple algorithms can already give optimal results, at least asymptotically. If, for example, in each step a random matching of H's edges is chosen and some load is sent via these edges if the loads of the corresponding processors are not balanced, then it is shown that the load deviation halves in a (up to a constant) minimal number of steps [30]. Unfortunately, such simple schemes are not really applicable in praxis. If the algorithm is supposed to balance the load locally such that the difference between neighboring processors is small, it needs an additional number of $\Omega(\sqrt{n})$ steps.

4.2.4 Diffusion

Cybenko and Boillat were the first to suggest a simple distributed load balancing strategy which they called *diffusion based load balancing* [10, 15]. In the diffusive load balancing algorithm each processor balances its load with all its neighbors in each round. These rounds are iterated until the load is completely balanced. Let w_p^t be the load of processor $p \in U$ in step t and let α_{pq} be a weighing factor on edge $\{p, q\}$ (e.g. $\alpha_{pq} = 1/deg_{\max}$). Then

$$w_p^{t+1} = w_p^t - \sum_{q,\ \{p,q\} \in F} \alpha_{pq}(w_p^t - w_q^t) \tag{7}$$

which means that the amount of $l_{pq}^{t+1} := \alpha_{pq}(w_p^t - w_q^t)$ load items is sent via edge $\{p,q\}$ between step t and $t+1$. Equation (7) can be written in matrix-vector notation. Let $w^t = (w_p^t)$ be the *load vector* and let $M = (\alpha_{pq})$ with $0 \leq \alpha_{pq} \leq 1$, $\alpha_{pq} = 0 \; \forall \{p,q\} \notin F$ and $\alpha_{pp} = 1 - \sum_{q \neq p} \alpha_{pq}$ be the *diffusion matrix*. Then (7) is equivalent to

$$w^{t+1} = M\, w^t \; . \tag{8}$$

The fact that M is doubly stochastic can be used to show that under certain additional conditions this iteration converges to an equally balanced load [15, 80]. The speed of convergence depends on the spectrum of M, especially on its second largest eigenvalue γ, and is determined by the structure of the graph F and the values of the α_{pq}. For certain graphs like hypercubes, tori or star graphs, optimal values of α_{pq} are known [23]. It can be shown that the standard diffusion algorithm as given in (8) reduces the load deviation to ϵ in $O(\frac{\log(1/\epsilon)}{1-\gamma^2})$ steps [31].

The relatively slow convergence of (8) can be improved by *over-relaxation* [80]. The idea is to monitor the amount of load sent over the edges and to determine the number of items to send in step t not only depending on the current load difference but also on the amount sent in the previous time step [31]. Iteration (8) changes to

$$w^1 = M\, w^0; \quad w^{t+1} = \beta M w^t + (1-\beta) w^{t-1} \; . \tag{9}$$

This can be turned into a local instruction. With l_{pq}^t being the amount of load moved over edge $\{p,q\}$ towards step t, (9) gives:

$$l_{pq}^{t+1} \;=\; \begin{cases} \alpha_{pq}(w_p^t - w_q^t) & \text{if } \; t = 0 \\ (\beta - 1) l_{pq}^t + \beta \alpha_{pq}(w_p^t - w_q^t) & \text{if } \; t > 0 \; . \end{cases} \tag{10}$$

It can be shown that the over-relaxed diffusion balances the load up to ϵ in $O(\frac{\log(1/\epsilon)}{\sqrt{1-\gamma^2}})$ steps if $\beta = \frac{2}{1+\sqrt{1-\gamma^2}}$ where γ is the second largest eigenvalue of M (according to amount) [31].

One of the main difficulties with the diffusion-type algorithms is the choice of good values for α and β. This implies not only a choice for α_{pq} minimizing γ. Additionally, γ has to be known in order to choose β. For a given graph F the best α_{pq} values can be determined using semi-definite programming (SDP) [23]. Unfortunately, SDP is very time- and space-consuming. Currently, optimal M's can only be determined for small graphs with less than 100 nodes. But the task of finding good parameters can also be reduced to an eigenvalue-problem of the corresponding Laplacian. The *Laplacian matrix* $L = D - A$ has already been defined in Section 3.3.1. If $\alpha_{pq} = \alpha$ for all $\{p,q\} \in F$ and L is the Laplacian matrix to H, then $M = I - \alpha \cdot L$. Thus, $\lambda_i(M) = 1 - \alpha\lambda_i(L)$ are the eigenvalues of M and $\alpha = \frac{2}{\lambda_{n-1}(L)+\lambda_1(L)}$ minimizes $\gamma(M)$ [23]. Additionally, γ and, hence, β are known. The eigenvalues of L can be estimated using the degree of the graph and an approximation of its edge-expansion. The latter can be determined using BFS starting from each node. This heuristic choice of parameters performs well on a number of graphs, is parallelizable and avoids the expensive calculation of eigenvalues of the graph [23].

4.2.5 Dimension Exchange

In the diffusion algorithm, a processor balances its load with *all* its neighbors in each step. If the processors are only allowed to communicate with one of their neighbors per step (i.e. the balancing is sequentialized), the convergence can be sped up by introducing sub-steps and using the information from the previous sub-step for balancing (instead of that from the beginning of the time-step) [87]. This method is called *Dimension Exchange* [15]. In one step (or round, sometimes also called *sweep* [88]) of this algorithm a processor still balances its load with all its neighbors but for the balancing over one edge (a sub-step) the most-up-to-date information is used:

$$w_p^{t+\Delta} = (1-\lambda)w_p^t + \lambda w_q^t \qquad \text{with} \quad \Delta = 1/deg_p \tag{11}$$

It can be easily seen that if H is a hypercube, one sweep with $\lambda = \frac{1}{2}$ is sufficient [15]. For certain other networks, optimal values of λ are known [88, 89].

4.2.6 Multilevel and Variants

The diffusion and dimension exchange algorithms show similarities to simple iterative methods for solving linear equational systems. The diffusion is a variant of the Jacobi iteration and dimension exchange may be viewed as a parallel version of Gauß-Seidel. In linear algebra, a number of other iterative methods for solving equational systems are known, many of which are much more efficient than Jacobi or Gau-Seidel. If we restrict to the model that a processor can only communicate with its neighbors and that no global knowledge is to be used, then diffusion and dimension exchange seem to be the only possible iterative methods which are applicable to the "Re-Embedding Problem" [31].

For practical applications such a restriction may not be valid, especially if the number of processors is not too large. We already saw that H might be considered as being a complete graph. Thus, communication between any pair of processors is possible. The problem still remains to determine how much load to shift over which edge of the quotient graph. But the processors may be allowed to communicate in a different way to calculate a solution to the flow problem.

A number of authors have studied multi-level methods for the "Re-Embedding Problem" [37, 76, 81]. They divide the quotient graph H into halves, determine the amount of load to shift across the cut and balance recursively in both halves [37, 81]. A variant reducing the total amount of load to be shifted divides the processors according to their current load, i.e. it applies a weighted partitioning of the quotient graph [76].

The co-ordination of balancing steps across a cut is a complex task. Nevertheless, the algorithms show a good performance in applications.

4.2.7 Minimal Flow

Hu and Blake formulate the local flow condition at nodes as an equational system [40]. For a node p it has to be

$$\sum_{q,\{p,q\}\in F} l_{pq} = w_p - \bar{w} \ , \tag{12}$$

i.e. the sum of in- and out-flow on its edges has to be equal to the difference between the current and the desired load value. The system (12) is under-determined (if H contains cycles) and, thus, has many solutions. Using Lagrange multipliers, Hu and Blake are able to minimize the Euclidean norm of the flow l over all edges. The system (12) turns into $L\lambda = b$ where λ is the vector of Lagrange multipliers and b is the right hand side of (12). The new system can be solved for λ using any equational solver. The flow values l_{pq} on edges $\{p,q\} \in F$ are then simply given by $l_{pq} = \lambda_p - \lambda_q$ [40].

Despite the fact that $\bar{w}$ is needed for this method, different equational solvers might increase the amount of global communication. Hu and Blake propose a CG algorithm for the solution of $L\lambda = b$ which implies a large number of scalar products, i.e. global sums. Nevertheless, on moderately sized systems the method seems to be fast and has the additional advantage of minimizing the amount of load movement. It develops to become the method of choice for load balancing in a number of applications [72, 83].

4.2.8 Information Dissemination

A drawback of many of the aforementioned methods is their relatively large run-time. On grids for example, the iterative methods need $O(P)$ steps in the worst case, if P is the number of processors [40]. The multi-level methods determine a balancing flow in $O(\log P)$ steps which is the best possible, but they are sometimes difficult to handle and their steps are very complex.

Algorithms from the field of information dissemination, especially *gossip algorithms* [57] may also be used to balance load. The gossip problem is such that each processor possesses a piece of information and that an optimal communication schedule is to be found ensuring that every processor knows the complete information afterwards. If this flow of information takes place on disjoint paths, then the same paths may be used for a flow of data-items to balance load.

Gossip problems in different communication modes are well studied in theoretical computer science [39, 57, 59]. See [46] for a survey.

4.3 "Which"

4.3.1 The General Problem

For the problem of "which", much less is known in literature. Here the task is to choose load items which can be migrated in order to fulfill the flow requirements determined in the "how

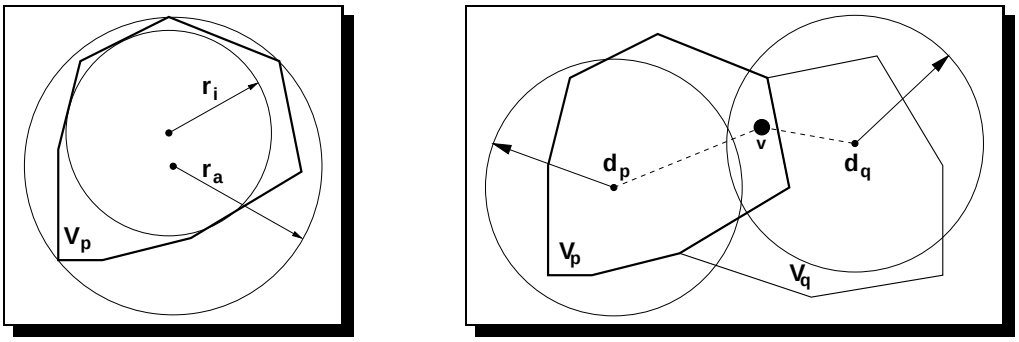

Figure 19: Aspect Ratio and approximation.

much"-step. The way such a choice is done depends on the application. If, for example, global iterative methods for solving linear systems like Multi-Grid or Conjugate Gradient are supposed to be parallelized, it is sufficient to choose load items in such a way that the communication demands are minimized. Measures of importance can be the total length of subdomain boundaries, the variance in boundary-length between all pairs of neighbors or the maximum degree of the cluster graph [21]. The details depend on the communication characteristics of the parallel system, especially on the values of overhead and latency (cf. Sec. 3.2). For simplicity, the total length of subdomain boundaries is considered in existing implementations of dynamic load balancing for adaptive FE calculations [83] (it should be pointed out that only few such implementations really exist currently).

The boundary length is a measure for the total amount of communication in each step of the algorithm. It is equivalent to the cut size. Thus, partitioning heuristics as described in Section 3 can be used (or at least adapted) to determine which load items to move.

Finite element meshes are usually partitioned along element boundaries. To be able to apply partitioning heuristics, the *element graph* can be used which contains a node for each element of the mesh. Edges in this graph express neighboring relations between elements. If we talk of moving load items in the following, this can either be elements or nodes of the element graph. We will use both notations synonymously.

4.3.2 The Special Case of DD-Preconditioning

Some applications like modern preconditioning techniques for finite element simulations [12] lead to additional requirements on the geometry of subdomains. The convergence characteristics of such preconditioned solvers is, among other things, determined by the shape of the partitions [79]. If, for example, domain decomposition techniques are used as preconditioning for conjugate gradient solvers [8], the ratio between the length of the longest and shortest "boundary edge" (which is a part of the domain boundary shared with a single neighbor) determines the convergence of the global iteration. On the other hand, the time needed for the local subdomain solutions is influenced by the "shape" of the subdomains, i.e. their ratio between area and diameter and whether they are convex or not. Any load migration strategy has to take these additional constraints into account in order

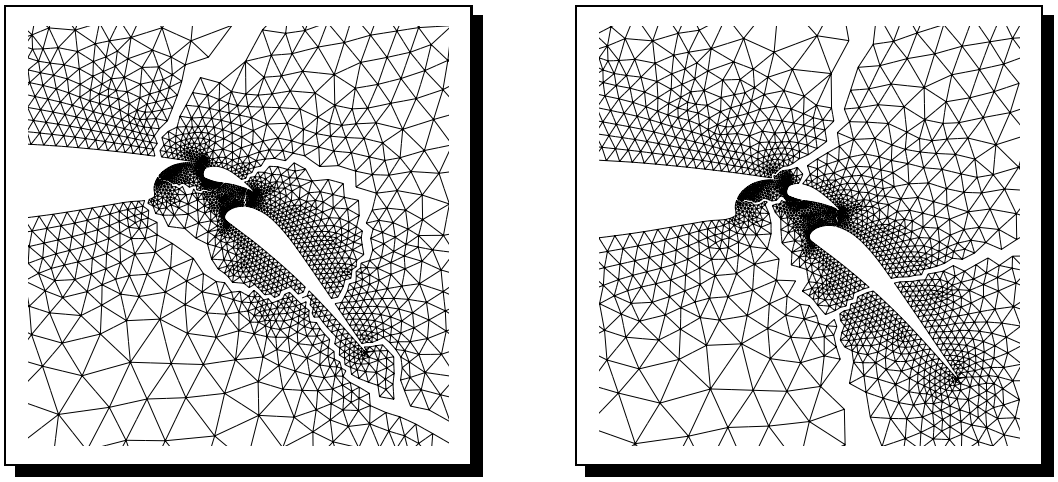

Figure 20: Partition generated by the spectral method (le), result after aspect-ratio optimization (ri).

to minimize the overall execution time.

A "shape optimizing" load migration is a complex task where currently only some very simple heuristics are known [13, 21]. They aim at optimizing the *aspect ratio* of subdomains. Figure 19 shows the definition of aspect ratio. It is given by the ratio between the radius of the largest inscribed circle to the radius of the smallest surrounding circle of a subdomain. Especially the inscribed circle is hard to maintain during a movement of elements. Thus, the existing heuristics in [13, 21] use approximations as described in Figure 19. They define the *radius* of a subdomain D_p mapped to processor $p \in U$ as $r_p = \sqrt{|D_p|}/\pi$ (in the 2D-case), approximating its area by the number of elements. The idea to optimize the shape is based on a migration of elements lying far away from the center of gravity of their subdomain. The distance of an element from the center of its cluster is normalized by the radius of the corresponding subdomain [13]. The change in shape is then set to the changing value of this measure for the originating and the destinating cluster of an element which is supposed to be moved:

$$f(v, D_q) = \frac{d(\delta_p, v)}{r_p} - 1 + \frac{d(\delta_q, v)}{r_q} - 1 \tag{13}$$

In equation (13), $d(v, w)$ is the Euclidean distance between two nodes and δ_p is the center of gravity of cluster p. If elements are to be moved from cluster p to cluster q, equation (13) is used to determine the specific elements [21].

Although $f(v, D_q)$ is only an approximation to the real aspect ratio, it improves this measure by up to 40% compared to a movement only accounting for cut size. Figure 20 shows an example of a zoom into a partition of the mesh "airfoil1". The partition on the left is calculated by spectral bisection, the right one is determined from an unbalanced partition using dimension exchange to determine "how much" and equation (13) for the "which" [21].

5 Conclusions

We gave a review of static and dynamic load balancing for applications from the field of scientific computing. We showed that both kinds of balancing problems can be modeled as graph embedding tasks.

The static problem of mapping a graph originating from such application (i.e. a mesh) onto a distributed system can be considered as being solved, at least for the majority of applications. Very efficient partitioning libraries allow the easy parallelization of FE-codes, even if the mesh is very large.

If the application is dynamic, i.e. if the mesh is adapted to the solution during runtime, a number of load balancing algorithms exist to determine how much load to move in which direction of the processor network in order to preserve load balance. Again, for the majority of applications the load balancing problem can be considered as being solved. But as there are currently only very few existing parallel implementations of adaptive codes, the balancing algorithms were not yet able to show their advantages or disadvantages.

If the solvers used in scientific codes are very complex, additional problems arise which complicate the static as well as the dynamic load balancing task. Especially for DD-preconditioning, additional constraints on the shape of subdomains need extra effort in partitioning and load balancing. Here, only very simple heuristics currently exist.

Acknowledgments

The authors would like to thank Petra Berenbrink, Stephan Blazy, Wolfgang Borchers, Thomas Decker, Uwe Dralle, Reinhard Lüling, S. Muthukrishnan, Markus Röttger and Chris Walshaw for many helpful discussions. Some of the sample graphs were provided by Steven Hammond, Alex Pothen, Jürgen Schulze and Horst Simon. Derk Meyer produced the software to generate some of the pictures. Chaco is used under license of Sandia Nat'l Lab. and is provided by Bruce Hendrickson and Robert Leland. Metis is due to George Karypis and Vipin Kumar.

References

[1] M. Adler, S. Chakrabarti, M. Mitzenmacher, L. Rasmussen: *Parallel randomized Load Balancing.* Proc. 27th ACM Symp. on Theory of Computing (STOC), 1995.

[2] Y. Azar, A.Z. Broder, A.R. Karlin, E. Upfal: *Balanced Allocations.* Proc. 26th ACM Symp. on Theory of Computing (STOC), 1994.

[3] A. Bäumker, W. Dittrich: *Fully dynamic search trees for an extension of the BSP model.* Proc. 8th ACM Symp. Parallel Algorithms and Architectures (SPAA '96), ACM Press, 233-242, 1996.

[4] S.T. Barnard, H.D. Simon: *Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems.* Concurrency - Practice and Exp. 6(2), 101-117, 1994.

[5] P. Bastian, S. Lang, K. Eckstein: *Parallel Adaptive Multigrid Methods in Structural Mechanics.* Num. Lin. Algebra with Appl., (to appear).

[6] L. Becchetti, A. Marchetti-Spaccamela, M. Röttger, U.-P. Schroeder, W. Unger: *An On-line Heuristic to Embed 2-Dimensional Refined Grids.* Techn. Rep., CS-Dept., Univ. of Paderborn, 1997.

[7] P. Berenbrink, F. Meyer a.d. Heide, K. Schröder: *Allocating Weighted Jobs in Parallel.* Proc. 9th ACM Symp. Parallel Algorithms and Architectures (SPAA), ACM Press, 302-310, 1997.

[8] S. Blazy, W. Borchers, U. Dralle: *Parallelization methods for a characteristic's pressure correction scheme.* Techn. Rep. tr-rsfb-96-028, Paderborn Univ. 1996.

[9] R. Blumofe, C.E. Leiserson: *Scheduling Multithreaded Computations by Work Stealing.* Proc. 35th Annual IEEE Symp. on Foundations of Computer Science (FOCS '94), 356-368, 1994.

[10] J.E. Boillat: *Load Balancing and Poisson Equation in a Graph.* Concurrency - Practice and Experience 2(4), 289-313, 1990.

[11] F. Bornemann, B. Erdmann, R. Kornhuber: *Adaptive Multilevel Methods in Three Space Dimensions.* Int. J. Num. Meth. Eng. (36), 3187-3203, 1993.

[12] J.H. Bramble, J.E. Pasciak, J. Xu: *Parallel Multilevel Preconditioners.* Mathematical Compututation, 55, 1-22, 1990.

[13] N. Chrisochoides, C.E. Houstis, E.N. Houstis, S.K. Kortesis, J.R. Rice: *Automatic Load Balanced Partitioning Strategies for PDE Computations.* Proc. ACM Int. Conf. on Supercomputing, 99-107, 1989.

[14] D.E. Culler, R.M. Karp, D.A. Patterson, A. Sahay, K.E. Schauser, E. Santos, R. Subronomian, T. van Eicken: *LogP: Towards a Realistic Model of Parallel Computation.* Proc. of 4th ACM SIGPLAN Symp. on Pinciples and Pract. of Par. Programming (PPoPP), 1993.

[15] G. Cybenko: *Load Balancing for Distributed Memory Multiprocessors.* J. of Parallel and Distributed Computing (7), pp. 279-301, 1989.

[16] T. Decker, R. Diekmann, R. Lüling, B. Monien: *Towards Developing Universal Dynamic Mapping Algorithms.* 17th IEEE Symp. on Parallel and Distributed Processing (SPDP '95), 456-459, 1995.

[17] R. Diekmann, U. Dralle, F. Neugebauer, T. Römke: *PadFEM: A Portable Parallel FEM-Tool.* HPCN-Europe, Springer LNCS 1067, 580-585, 1996.

[18] R. Diekmann, R. Lüling, B. Monien, C. Spräner: *Combining Helpful Sets and Parallel Simulated Annealing for the Graph-Partitioning Problem.* Parallel Algorithms and Applications 8, 61-84, 1996.

[19] R. Diekmann, R. Lüling, A. Reinefeld: *Distributed Combinatorial Optimization.* Proc. of Sofsem'93, Czech Republik, pp. 33–60, 1993.
`http://www.uni-paderborn.de/cs/diek.html`

[20] R. Diekmann, R. Lüling, J. Simon: *Problem Independent Distributed Simulated Annealing and its Applications.* In R.V.V. Vidal (ed): *Applied Simulated Annealing*, Lect. Notes in Ec. and Math. Systems, Springer LNEMS 396, pp. 17-44, 1993.

[21] R. Diekmann, D. Meyer, B. Monien: *Parallel Decomposition of Unstructured FEM-Meshes.* Proc. IRREGULAR '95, Springer LNCS 980, 199-215, 1995. Concurrency - Practice and Experience, 1997.

[22] R. Diekmann, B. Monien, R. Preis: *Using Helpful Sets to Improve Graph Bisections.* Hsu et.al. (ed.): *Interconnection Networks and Mapping...* DIMACS Discr. Math. and Th. CS (21), AMS, 57-73, 1995.

[23] R. Diekmann, S. Muthukrishnan, M.V. Nayakkankuppam: *Engineering Diffusive Load Balancing Algorithms Using Experiments.* Proc. IRREGULAR '97, Springer LNCS 1253, 111-122, 1997.

[24] C. Farhat: *A Simple and Efficient Automatic FEM Domain Decomposer.* Computers & Structures 28(5), 579-602, 1988.

[25] C. Farhat, S. Lanteri, H.D. Simon: *TOP/DOMDEC - a Software Tool for Mesh Partitioning and Parallel Processing.* J. Computing Systems in Engineering 6(1), 13-26, 1995.

[26] S. Felperin, P. Raghavan, E. Upfal: *A Theory of Wormhole Routing in Parallel Computers.* ACM Symp. Foundat. of Comp. Sc. (FOCS '92), 563-572, 1992.

[27] M. Fiedler: *Algebraic Connectivity of Graphs.* Czechoslovak Math. J. 23, 298-305, 1973.

[28] C.M. Fiduccia, R.M. Mattheyses: *A linear-time heuristic for improving network partitions.* Proc. 19th IEEE Design Automation Conference, 175-181, 1982.

[29] M.R. Garey, D.S. Johnson, L. Stockmeyer: *Some Simplified* NP*-complete Graph Problems.* Theoretical Computer Science 1, 237-267, 1976.

[30] B. Ghosh, F.T. Leighton, B.M. Maggs, S. Muthukrishnan, C.G. Plaxton, R. Rajaraman, A. Richa, R.E. Tarjan, D. Zuckerman: *Tight Analyses of Two Local Load Balancing Algorithms.* Proc. 27th ACM Symp. on Theory of Computing (STOC '95), 548-558, 1995.

[31] B. Ghosh, S. Muthukrishnan, M.H. Schultz: *First and Second Order Diffusive Methods for Rapid, Coarse, Distributed Load Balancing.* Proc. 8th ACM Symp. Parallel Algorithms and Architectures (SPAA '96), ACM Press, 72-81, 1996.

[32] H.-U. Heiß, M. Dormanns: *Partitioning and Mapping of Parallel Programs by Self Organization.* Concurrency - Practice and Experience (CPE), 1996.

[33] B. Hendrickson, R. Leland: *The Chaco User's Guide.* Techn. Rep. SAND94-2692, Sandia Nat'l Lab, 1994.

[34] B. Hendrickson, R. Leland: *A Multilevel Algorithm for Partitioning Graphs.* Techn. Report SAND93-1301, Sandia Nat'l Lab, Oct. 1993.

[35] B. Hendrickson, R. Leland: *An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations.* SIAM J. Scientific Computing 16(2), 452-469, 1995.

[36] V. Heun, E.W. Mayr: *Efficient Dynamic Embedding of Arbitrary Binary Trees into Hypercubes.* Proc. 3rd Int'l Symp. on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR'96), Springer LNCS 1117, 287-298, 1996.

[37] G. Horton: *A multi-level diffusion method for dynamic load balancing.* Parallel Computing 19 (1993), pp. 209-218

[38] J. Hromkovič, B. Monien: *The Bisection Problem for Graphs of Degree 4 (Configuring Transputer Systems).* In Buchmann-Ganzinger-Paul, (Festschrift zum 60. Geburtstag von Günter Hotz), B.G. Teubner, Stuttgart-Leipzig, 215-234, 1992.

[39] J. Hromkovic, R. Klasing, W. Unger, H. Wagener: *Optimal Algorithms for Broadcast and Gossip in the Edge-Disjoint Path Modes.* Proc. 4th Scandinavian Workshop on Algorithm Theory (SWAT'94), Springer LNCS 824, 219-230, 1994.

[40] Y.F. Hu, R.J. Blake: *An optimal dynamic load balancing algorithm.* Techn. Rep. DL-P-95-011, Daresbury Lab., UK, 1995. (to appear in: Concurrency - Practice and Experience)

[41] M.T. Jones, P.E. Plassmann: *Parallel Algorithms for the Adaptive Refinement and Partitioning of Unstructured Meshes.* Proc. Scalable High Performance Computing Conference, IEEE Computer Society Press, 478-485, 1994.

[42] M. Kaddoura, C.-W. Ou, S. Ranka: *Mapping unstructured Computational Graphs for Adaptive and Nonumiform Computational Environments.* IEEE Par. and Distr. Technology, Sept. 1995.

[43] G. Karypis, V. Kumar: *A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs.* Techn. Rep. 95-035, CS-Dept., Univ. Minnesota, 1995.

[44] G. Karypis, V. Kumar: *A Coarse-Grain Parallel Formulation of a Multilevel k-way Graph Partitioning Algorithm.* Proc. 8th SIAM Conf. on Parallel Processing for Scientific Computing, 1997.

[45] B.W. Kernighan, S. Lin: *An Effective Heuristic Procedure for Partitioning Graphs.* The Bell Systems Technical Journal, 291-308, Feb. 1970.

[46] R. Klasing: *On the Complexity of Broadcast and Gossip in Different Communication Modes.* Shaker Verlag, Aachen, Germany, 1996.

[47] F.T. Leighton: *Introduction to Parallel Algorithms and Architectures.* Morgan Kaufmann Publishers, 1992.

[48] T. Leighton, M. Newman, A. Ranade, E. Schwabe: *Dynamic Tree Embedding in Butterflies and Hypercubes.* SIAM Journal on Computing 21(4), 639-654, August 1992.

[49] T. Lengauer: *Combinatorial Algorithms for Integrated Circuit Layout.* B.G. Teubner, 1990

[50] F.C.H. Lin, R.M. Keller: *The Gradient Model Load Balancing Method.* IEEE Trans. on Software Engineering 13, 32-38, 1987.

[51] R. Lüling, B. Monien: *Load Balancing for Distributed Branch & Bound Algorithms.* Proc. 6th Int'l Parallel Processing Symp. (IPPS '92), 543-549, 1992.

[52] R. Lüling, B. Monien: *A Dynamic Distributed Load Balancing Algorithm with Provable Good Performance.* Proc. 5th Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA '93), ACM Press, 164-173, 1993.

[53] R. Lüling, B. Monien, F. Ramme: *Load Balancing in Large Networks: A Comparative Study.* Proc. 3rd IEEE Symp. Parallel and Distributed Processing (SPDP '91), 686-689, 1991.

[54] B.M. Maggs, L.R. Matheson, R.E. Tarjan: *Models of Parallel Computation: A Survey and Synthesis.* Proc. 28th Hawaii Int'l Conf. on System Sciences (HICSS-28) Vol. 2, 61-70, 1995.

[55] F. Meyer auf der Heide, B. Oesterdiekhoff, R. Wanka: *Strongly Adaptive Token Distribution.* Algorithmica 15, 413-427, 1996.

[56] B. Monien, R. Diekmann: *A Local Graph Partitioning Heuristic Meeting Bisection Bounds.* Proc. 8th SIAM Conf. on Parallel Processing for Scientific Computing, 1997.

[57] B. Monien, R. Diekmann, R. Feldmann, R. Klasing, R. Lüling, K. Menzel, T. Römke, U.-P. Schroeder: *Efficient Use of Parallel & Distributed Systems: From Theory to Practice.* In: J. van Leeuwen (ed.): *Computer Science Today*, Springer LNCS 1000, 62-77, 1995.

[58] B. Monien, R. Diekmann, R. Lüling: *Communication Throughput of Interconnection Networks.* Proc. 19th Int'l Symp. on Mathematical Foundations of Computer Science (MFCS '94), Springer LNCS 841, 72-86, 1994.

[59] B. Monien, R. Feldmann, R. Klasing, R. Lüling: *Parallel Architectures: Design and Efficient Use.* 10th Symp. Theor. Aspects of Computer Science (STACS '93), Springer LNCS 665, 1993.

[60] B. Monien, I.H. Sudborough: *Embedding One Interconnection Network in Another.* Computing Suppl. 7, 257-282, 1990.

[61] S.R. Öhring, S.K. Das: *Mapping Dynamic Data and Algorithm Structures into Product Networks.* In: K.W. Ng (ed.): *Algorithms and Computation.* Proceedings, Springer LNCS 762, 147-156, 1993.

[62] L. Oliker, R. Biswas: *Efficient Load Balancing and Data Remapping for Adaptive Grid Calculations.* Proc. 9th ACM Symp. Parallel Algorithms and Architectures (SPAA '97), ACM Press, 33-42, 1997.

[63] F. Pellegrini and J. Roman: *Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs.* HPCN-Europe, Springer LNCS 1067, 493-498, 1996.

[64] A. Pothen, H.D. Simon, K.P. Liu: *Partitioning Sparse Matrices with Eigenvectors of Graphs.* SIAM J. Matr. Anal. Appl. 11/3, 430-452, 1990.

[65] R. Preis, R. Diekmann: *The PARTY Partitioning-Library, User Guide.* Techn. Rep. tr-rsfb-96-024, CS-Dept., Univ. of Paderborn, 1996.

[66] R. Preis, R. Diekmann: *PARTY - A Software Library for Graph Partitioning.* In: B.H.V. Topping (ed.): *Advances in Computational Mechanics ...*, Civil-Comp Press, Edinburgh, 63-71, 1997.

[67] A. Ranade: *Optimal Speedup for Backtrack Search on a Butterfly Network.* Proc. 3rd Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA '91), ACM Press, 40-48, 1991.

[68] R. Rehrmann, B. Monien, R. Lüling, R. Diekmann: *On the Communication Throughput of Multistage Interconnection Networks.* Proc. 8th ACM Symp. Parallel Algorithms and Architectures (SPAA '96), ACM Press, 152-161, 1996.

[69] T. Römke, M. Röttger, U.-P. Schroeder, J. Simon: *On Efficient Embeddings of Grids into Grids in PARIX.* Proc. 1st Int'l EURO-PAR Conference, Springer LNCS 966, 181-204, 1995.

[70] M. Röttger, U.-P. Schroeder, J. Simon: *Implementation of a Parallel and Distributed Mapping Kernel for PARIX.* Proc. HPCN Europe '95, Springer LNCS 919, 781-786, 1995.

[71] L. Rudolph, M. Slivkin-Allalouf, E. Upfal: *A Simple Load Balancing Scheme for Task Allocation in Parallel Machines.* Proc. 3rd Annual ACM Symp. on Parallel Algorithms and Architectures (SPAA '91), ACM Press, 237-245, 1991.

[72] K. Schloegel, G. Karypis, V. Kumar: *Repartitioning of Adaptive Meshes: Experiments with Multilevel Diffusion.* Report, CS-Dept., UMN, 1997.

[73] H.D. Simon: *Partitioning of Unstructured Problems for Parallel Processing.* Computing Systems in Engineering (2), 135-148, 1991.

[74] H.D. Simon, A. Sohn, R. Biswas: *HARP: A Fast Spectral Partitioner.* Proc. 9th ACM Symp. Parallel Algorithms and Architectures (SPAA '97), ACM Press, 43-52, 1997.

[75] A. Sohn, R. Biswas, H.D. Simon: *A Dynamic Load Balancing Framework for Unstructured Adaptive Computations on Distributed-Memory Multiprocessors.* Proc. 8th ACM Symp. Parallel Algorithms and Architectures (SPAA '96), ACM Press, 189-192, 1996.

[76] N. Touheed, P.K. Jimack: *Parallel Dynamic Load-Balancing for Adaptive Distributed Memory PDE Solvers.* Proc. 8th SIAM Conf. on Parallel Processing for Scientific Computing, 1997.

[77] S. Tschöke, R. Lüling, B. Monien: *Solving the Traveling Salesman Problem with a Distributed Branch-and-Bound Algorithm on a 1024 Processor Network.* Proc. 9th Int. Parallel Processing Symp. (IPPS '95), 182-189, 1995.

[78] L. Valiant: *A Bridging Model for Parallel Computations.* Communications of the ACM, Vol. 33, No. 8, Aug. 1994.

[79] D. Vanderstraeten, R. Keunings: *Optimized Partitioning of Unstructured Computational Grids.* Int. J. Num. Meth. Eng., 38, 433-450, 1995.

[80] R. Varga: *Matrix Iterative Analysis.* Prentice-Hall, 1962.

[81] A. Vidwans, Y. Kallinderis, V. Venkatakrishnan: *Parallel Dynamic Load Balancing Algorithm for Three-Dimensional Adaptive Unstructured Grids.* AIAA Journal, 32(3), 497-505, 1994.

[82] C. Walshaw, M. Berzins: *Dynamic load-balancing for PDE solvers on adaptive unstructured meshes.* Concurrency - Practice and Experience 7(1), 1995.

[83] C. Walshaw, M. Cross, M. Everett: *Dynamic load-balancing for parallel adaptive unstructured meshes.* Proc. 8th SIAM Conf. on Parallel Processing for Scientific Computing, 1997.

[84] R.D. Williams: *Unification of Spectral and Inertial Bisection.* Techn. Report, CalTech, 1994. `http://www.cacr.caltech.edu/~roy/papers/`

[85] R.D. Williams: *Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations.* Concurrency 3, 457, 1991.

[86] C.-Z. Xu, R. Diekmann, F.C.M. Lau: *Decentralized Remapping of Data Parallel Applications on Distributed Memory Multiprocessors.* Concurrency - Practice and Experience (CPE), to appear.
Also: Techn. Rep. No. tr-rsfb-96-021, University of Paderborn, 1996.

[87] C. Xu, B. Monien, R. Lüling, F. Lau: *Nearest-neighbor algorithms for load-balancing in parallel Computers.* Concurrency - Practice and Experience, Vol. 7(7), pp. 707–736, 1995.

[88] C. Xu, F. Lau: *The Generalized Dimension Exchange Method for Load Balancing in k-ary n-cubes and Variants.* J. Parallel and Distributed Computing 24(1), 72-85, 1995.

[89] C.-Z. Xu, F.C.M. Lau: *Load balancing in Parallel Computers: Theory and Practice.* Kluwer Academic Publishers, 1997.