Min-Cut Methods for Mapping Dataflow Graphs

Volker Elling[1] and Karsten Schwan[2]

[1] RWTH Aachen, Bärenstraße 5 (Apt. 12), 52064 Aachen, Germany
[2] Georgia Institute of Technology, 801 Atlantic Drive, Atlanta, GA 30332-0280, USA

Abstract. High performance applications and the underlying hardware platforms are becoming increasingly dynamic; runtime changes in the behavior of both are likely to result in inappropriate mappings of tasks to parallel machines during application execution. This fact is prompting new research on mapping and scheduling the dataflow graphs that represent parallel applications. In contrast to recent research which focuses on critical paths in dataflow graphs, this paper presents new mapping methods that compute near-min-cut partitions of the dataflow graph. Our methods deliver mappings that are an order of magnitude more efficient than those of DSC, a state-of-the-art critical-path algorithm, for sample high performance applications.

1 Introduction

Difficult steps in parallel programming include decomposing a computation into concurrently executable components ('tasks'), assigning them to processors ('mapping') and determining an order of execution on each processor ('scheduling'). When parallel programs are run on dedicated multiprocessors with known processor speeds, network topologies and bandwidths, programmers can use fixed task-to-processor mappings, develop good mappings by trial-and-error using performance analysis tools to identify bottlenecks, perhaps based on the output of parallelizing compilers. The difficulties with these approaches are well known. First, many problems, including 'sparse triangular solves' (see Section 4.3), are too irregular for partitioning by a human. Second, when programs are decomposed by compilers, at a fine grain of parallelism, the potential number of parallel tasks is too large for manual methods. Third, when using LAN-connected clusters of workstations (see Figure 1 for the Georgia Tech IHPCL lab) or entire computational grids (as in Globus [13]), there are irregular network topologies and changes in network or machine performance due to changes in load or platform failures. Fourth, task sets may change at runtime either due to the algorithm or because of computer-user interaction like visualization or steering [19, 15, 8]. Finally, in load balancing a busy processor occasionally shares tasks with a processor that has become idle. It is useful to choose the share of the latter processor so that the communication between both is minimized; this requires min-cut partitioning methods for dataflow graphs (see below). In all these situations, automatically generated mappings are attractive or even required. For these and other reasons, the ultimate goal of our research is to develop black-box mapping and scheduling packages that require little or no human intervention.

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 203–212, 1999.

In Section 2, we define the formal problem that is to be solved and present examples for its relevance. Section 3 discusses critical-path vs. min-cut mapping methods. Our algorithmic contributions are presented in Sections 3.3 and 3.4.

Section 4.1 introduces an abstract model of parallel hardware (suited for multiprocessor machines as well as for workstation clusters), which is then used for extending 2-processor mapping methods to an arbitrary number of processors. In Section 4.3, we present two real-world sample problems; simulation results for them are shown, for our two methods as well as DSC, a critical-path algorithm, and 'DSC-spectral', a variant of DSC. For more details about our algorithms and experiments see [9, 10].

2 Formal Problem Definition

Sample Applications. Parallel applications are often modelled using dataflow graphs (also known as 'macro dataflow models', 'task graphs', or 'program dependence graphs'). A dataflow graph is a directed acyclic graph (V, E) consisting of vertices $v \in V$ representing computation steps, connected by directed edges $(v, w) \in E$ which represent data generated by v and used by w. The vertices are weighted with the computational costs $t(v)$, whereas edges are weighted with the amounts of communication $c(v, w)$. We prefer cost measures like 'FLOPS' or 'bytes' rather than 'execution time' resp. 'latency' because the latter depend on where execution/communication take place which is a priori unknown. An example is the graph shown in Figure 1. This graph represents a two-hour iteration step in the Georgia Tech Climate Model (GTCM) which simulates atmospheric ozone depletion. On our UltraSparc-II-cluster, the tasks in the 'Lorenz iteration' execute for about 2 seconds while the chemistry tasks run for about 10 seconds. One execution of the task graph represents 2 hours simulated time; problem size ranges from 1 month (for 'debugging') to 6 years of simulated time. Each graph edge corresponds to 100–400 KB of data. This application constitutes one of the examples addressed by our mapping (and remapping) algorithm. Another example exhibiting finer grain parallelism is presented in Section 4.3 below.

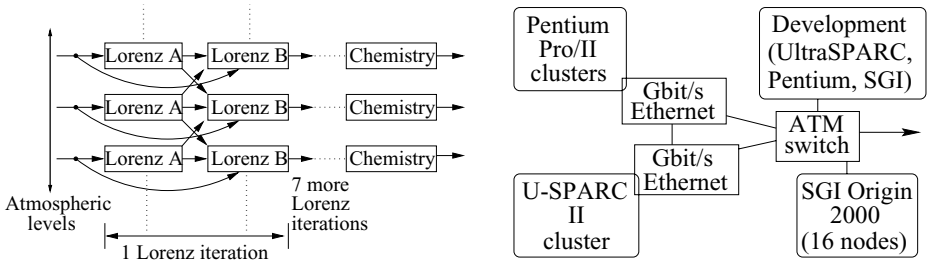

Fig. 1. Left: task graph in the Georgia Tech Climate Model; right: high-performance hardware in the Georgia Tech IHPCL lab

Problem Definition. 'Mapping' is the task of assigning each vertex v to $\pi(v)$, one of P available processors. 'Scheduling' (more precisely, 'local scheduling') is the subsequent stage of determining an order of execution for the vertices on each processor (this order can be strict or advisory). Usually, the objective is to minimize the 'makespan', the time between start of the first and completion of the last task. We define

$$\text{efficiency} = \frac{\sum_{v \in V} \text{load}_v}{\text{makespan} \cdot \sum_{p=1}^{P} \text{speed}_p}$$

where load_v refers to the number of t-units task v takes, and speed_p refers to the speed of processor p in t-units per second. The numerator contains the total workload measured in t-units, the denominator contains the amount of t-units provided by the processors during the makespan.

Many existing mapping/scheduling algorithms are restricted to homogeneous processors connected by a uniform network. For the cluster and grid machines used in our work (see Figure 1), however, we have to deal with processors of different speeds ('weakly heterogeneous multiprocessors') or even different architectures ('strongly heterogeneous multiprocessors' — some processors are well-suited for floating-point computations (MIPS, UltraSPARC) while others were designed for integer tasks (Intel)). The size of on-chip caches is important as well. Some machines (SGI Origin) utilize high performance interconnects, whereas the workstation clusters employ commodity networks like switched Ethernet for interprocessor communication. Finally, processor and network speeds can vary over time due to shared use with other applications. The methods we describe deal with weak heterogeneity, and they are suited for remapping in case of changes in program behavior or resource availability.

The performance of parallel applications is affected by various factors, including CPU performances, amounts of main memory and disk space, network latencies and bandwidths. Some parallel programs are bandwidth-limited in the sense that network congestion causes slowdown. Others are latency-limited: slowdown results from point-to-point delays in frequent, fine-grain communication. This paper considers both latency- and bandwidth-limited programs.

3 Mapping Algorithms

This section first describes a popular mapping algorithm called 'Dominant Sequence Clustering' (DSC). Next, we present new mapping algorithms, called 'spectral mapping' and 'greedy mapping'. In Section 4, these algorithms are shown to be superior to DSC in terms of mapping quality.

3.1 Critical-Path Methods

Critical-path methods require $t(v)$ resp. $c(v, w)$ to be computation resp. communication time. If the sum of $t(v)$ and $c(v, w)$ on a path in the dataflow graph (the 'path length') equals the makespan, the path is called 'critical'. In order to decrease the makespan, the lengths of all critical paths have to be decreased (by assigning v to a faster processor

resp. v, w to the same processor or processors connected by a faster link). The most advanced critical-path algorithm known to us is 'Dominant Sequence Clustering' ('DSC', [22]). It is as efficient as older algorithms in minimizing makespan [14]; in addition, it is faster since it recomputes critical paths less often.

Usually, critical-path methods do not try to minimize cut size. DSC forms clusters of tasks; the clusters are assigned to processors blockwise or cyclically. In [22], the use of Bokhari's algorithm [3] for overcoming this limitation is proposed: an undirected graph is formed, with the clusters as vertices and edges (c, d) weighted with the amount of communication $C(c, d)$ between clusters c and d:

$$C(c,d) := \sum_{v \in c, w \in d} (c(v,w) + c(w,v)).$$

The cut size is reduced by computing small-cut partitions of the cluster graph and assigning them to processors. We propose to partition the graph by spectral bisection rather than Bokhari's algorithm (below, this variant is referred to as 'DSC-spectral').

3.2 Min-Cut Methods

Min-cut methods do not proceed by shortening critical paths; they try to find a mapping $\pi : V \to \{1, \ldots, P\}$ of the task graph with small cut size, and loads approximately proportional to the respective speeds of the processors $p = 1, \ldots, P$:

$$\mathrm{cutsize}(\pi) := \sum_{(v,w) \in E,\ \pi(v) \neq \pi(w)} c(v,w), \qquad \mathrm{load}_p(\pi) := \sum_{\pi(v)=p} t(v).$$

Previous Work. To the best of our knowledge, there has been no previous attempt to define explicit min-cut mapping algorithms for arbitrary *dataflow* graphs. However, dataflow graphs often arise from undirected graphs like finite-element grids. There has been progress on min-cut partitioning algorithms for *undirected* graphs ([17, 12, 20, 16]; for a survey see [11]). Unfortunately, these methods cannot be trivially applied to the problem we are solving, since our complex parallel applications typically consist of several coupled subcomponents (e.g., a finite-element elasticity code, a finite-volume gas dynamics code, chemistry and visualisation) that cannot be scheduled easily by partitioning a single physical grid.

In [4], an undirected doubly-weighted graph of communicating long-running processes is partitioned in a Simulated Annealing fashion. [5] proposes a method based on greedy pairwise exchange, for a problem similar to our dataflow graphs. However, this method recomputes the makespan (or a similar objective function) after every exchange which is very expensive.

Min-cut algorithms for directed graphs cannot easily be adapted to dataflow graphs since the partition with smallest cut size might be the *worst* rather than the best mapping (see Figure 2). As evident from the example in the figure, it is important to take the directedness of the graph into account. Toward this end, we define the earliest-start resp. earliest-finish 'times' $\mathrm{est}(v)$ and $\mathrm{eft}(v)$ by

$$\mathrm{est}(v) := \max_{(w,v) \in E} \mathrm{eft}(w), \qquad \mathrm{eft}(v) := \mathrm{est}(v) + t(v)$$

(note that this definition is valid because the graph is acyclic). The nodes are sorted by est and separated into K sets $V_1, \ldots, V_K$ with (almost) equal size so that

$$i \le j, v \in V_i, w \in V_j \quad \Rightarrow \quad \mathrm{est}(v) \le \mathrm{est}(w).$$

Instead of requiring load proportional to speed on each processor, we require proportionality *in each set* V_k:

$$\mathrm{speed}_p \sim \mathrm{load}_{p,k}(\pi) := \sum_{v \in V_k,\ \pi(v)=p} t(v).$$

This means that we have load balance in K 'time intervals' rather than overall. Of course, the exact start and finish times of the tasks are not known in advance; in addition, our definition of est and eft does *not* involve $c(v, w)$. Nevertheless, est and eft are a practical way of estimating the relative execution order of tasks in advance. In our experiments, we compute the longest (here, length = number of vertices) path in the dataflow graph and set K to half its length. Figure 2 shows the improved mapping.

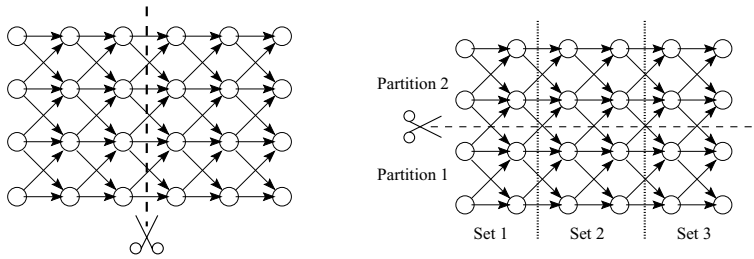

Fig. 2. Left: a bad min-cut mapping; right: improved mapping

3.3 Spectral Mapping

We have adapted spectral bisection[20] to dataflow graphs since it delivers, along with multilevel partitioning[16], the best partitions for undirected graphs. Its disadvantages are low speed and difficult implementation.

For simplicity, we assume that $V = \{1, \ldots, |V|\}$. The 'Laplacian matrix' (compare [7]) $L = (L_{vw})$ is defined by

$$L_{vw} := \begin{cases} -c(v,w), & v \ne w, (v,w) \in E \text{ or } (w,v) \in E \\ \sum_{(v,w)\in E} c(v,w), & v = w \\ 0, & \text{else} \end{cases}.$$

It is a positive semidefinite matrix, with $(1, \ldots, 1)$ as eigenvector for the eigenvalue 0. The second smallest eigenvalue is always positive if the graph is connected. The corresponding eigenvector x is called 'Fiedler vector'. It solves

$$\phi(x) := \frac{\sum_{v,w\in E} c(v,w)(x_v - x_w)^2}{\sum_{v\in E} x_v^2} \longrightarrow \min, \qquad \text{s.t. } \exists v, w : x_v \ne x_w$$

Note that $\phi(x)$ is small if, for adjacent vertices v, w, the difference $x_v - x_w$ is small. The 'closer' vertices are in the graph, the closer are their x-values. For dataflow graphs, we minimize $\phi(x)$ with respect to the constraints

$$\sum_{v \in V_k} t(v) x_v = 0.$$

This corresponds to finding the smallest eigenvalue and corresponding eigenvector of the operator $P^\top L P$ where P is an orthogonal $n \times (n-K)$ matrix mapping $\mathbb{R}^{n-K}$ into the constraint subspace of $\mathbb{R}^n$. A 'bisection' (2-partition) is obtained by choosing thresholds T_k, $k = 1, \ldots, K$, and setting, for $v \in V_k$,

$$\pi(v) := \begin{cases} 1, & x_v > T_k \\ 0, & \text{else} \end{cases}.$$

Let P_i be the set of processors executing partition i; the T_k are chosen so that

$$\frac{\mathrm{load}_{0,k}(\pi)}{\mathrm{load}_{0,k}(\pi) + \mathrm{load}_{1,k}(\pi)} \approx \alpha := \frac{\sum_{p \in P_0} \mathrm{speed}_p}{\sum_{p \in P_0 \cup P_1} \mathrm{speed}_p}$$

Spectral mapping is much slower than DSC or greedy mapping (see below), but it takes only about 1.7 seconds for a 1000 vertices/4000 edge graph (20 seconds for 10000/40000) on an SGI O2 (R10000 2.6 195 MHz). These times can be improved significantly, as our current implementation is sequential and not very sophisticated; [1, 2] propose multilevel parallelized variants for spectral bisection that achieve an order-of-magnitude performance improvement and can be adapted to spectral mapping.

3.4 Greedy Mapping

A faster but lower-quality method for min-cut bisection is 'greedy mapping'. It corresponds to the greedy bisection methods developed for undirected graphs[17, 12]. At the beginning, an arbitrary initial mapping π is chosen. The 'gain' of a vertex is defined as the decrease in cutsize when this vertex is moved to the other partition ($\pi(v)$ is changed from 0 to 1 or vice versa). The vertices are moved between partitions, in order of decreasing gain. In every iteration, a vertex may be moved only once. A vertex $v \in V_k$ may not be moved if moving it would make

$$|\alpha - \frac{\mathrm{load}_{0,k}(\pi)}{\mathrm{load}_{0,k}(\pi) + \mathrm{load}_{1,k}(\pi)}|$$

larger than a threshold (for example, 0.07). An iteration finishes when no vertices with nonnegative gain are left. In our experience, there is no improvement after 10–15 iterations.

4 Evaluation

4.1 Modeling Topologies

This section proposes a simple but representative model for real-world networks. A model consists of processors with different speed, each connected to an arbitrary number of buses. Buses themselves can be connected by switches ('zero-speed processors').

In order to apply our bisection algorithms to topologies with more than 2 processors, a hierarchy of 'nodes' is computed. At each step, the bus with highest bandwidth is chosen and, together with the connected processors (atomic nodes), folded into a single parent node. When all buses have been folded, only one node is left. The clustering is undone in reverse order. At each step, one node is unfolded into a bus and the adjacent nodes. The dataflow graph partition corresponding to the parent node is distributed to the children by repeated bisection. α is chosen according to processor performances in each bisection step. Finally, all steps have been undone, and every processor has been assigned a partition of the graph.

As an example, consider Figure 1. Each cluster is (somewhat inaccurately) modeled as a bus to which machines and switches are connected. The p-processor machines can be treated as single-processor machines with p-fold speed, or (for large multiprocessor machines) as a separate bus with p nodes connected to it. Starting with the Gigabit Ethernet links, each link is folded into a node. After this, the ATM links are folded. Our bisection algorithm will try to assign coherent pieces of the dataflow graph to the Gigabit clusters and minimize communication via the slow ATM switch. However, the example also demonstrates the limitations of our simplistic topology treatment: the link 'A' interconnecting the two Gigabit switches might be folded first (before any of the Gigabit-switch-to-processor links is folded). Since it might represent a bottleneck, folding it *last* (i.e. assigning coherent pieces of the dataflow graph to the processors on each side) is more appropriate. A more sophisticated algorithm would consider the cluster topology as an undirected graph and apply spectral bisection to it.

4.2 Local Scheduling

[21] discusses good heuristics for computing local schedules on processors and compares them for randomly generated graphs. Interestingly, taking communication delays into account (as in the 'RCP*' scheme) and neglecting them (in the 'RCP' scheme) does not affect schedule quality. In our experiments we use the RCP scheme (in its non-strict form, i.e. out-of-order execution is allowed if the highest-priority task is not ready).

4.3 Sample Problems

We have chosen two sample problems that are rather complementary and reflect the variety of parallel applications. The first, 'sparse triangular solves', is very fine-grain and latency-limited: each task runs for $\leq 1\ \mu s$, each edge corresponds to 12 bytes. The second problem, the Georgia Tech climate model[18] which has already been introduced in Section 2, is a very coarse-grain and usually bandwidth-limited problem. The tasks execute for several seconds; the edges correspond to data in the order of 100 kilobytes. The model runs for a very long time; this is typical for many of the so-called grand-challenge applications.

In our simulations, we use some simplifications: data is sent in packets that have equal size on all buses. The bandwidth of a bus is determined by the number of packets per second. Network interfaces have unlimited memory and perfect knowledge about

$$\begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ -1 & 2 & 0 & 0 & 0 \\ 2 & 0 & 1 & 0 & 0 \\ 0 & 1 & -1 & 1 & 0 \\ 1 & 3 & 0 & -1 & 1 \end{pmatrix}$$

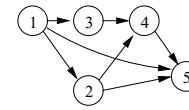

Fig. 3. Lower triangular matrix and its dataflow graph. Vertices are labeled with the index of the corresponding matrix row.

the other interfaces on the bus. When a bus becomes available, one of the waiting interfaces is chosen randomly with uniform probability. These simplifications are not vital. By varying the packet size it is possible to simulate networks with different latencies.

Sparse Triangular Solves. Our first test problem are sparse triangular solves (STS): solve for x in the linear system $Ax = b$ where A is a lower triangular matrix with few nonzero entries. Task i corresponds to solving for x_i; this requires all x_j with $a_{ij} \neq 0$. The dataflow graph is determined by the sparsity structure of A (see Figure 3). The time for distribution of A, b is neglected (initial data locations depend on the application).

A workstation cluster is appropriate only for very large A; otherwise a multiprocessor machine with good interconnect is mandatory. Results from [6] for a CM-5 implementation of STS with mapping by DSC suggest that DSC is too slow: mapping time exceeds execution time, even if one mapping is reused many times (which is possible in typical applications). Since spectral mapping is slower than DSC, we consider STS as a source for real-world dataflow graphs rather than a practical application. The results shown in Figure 4 were obtained by simulation using the hardware model in Section 4.1, with $t(v) = 1\ \mu s$, $c(v, w) = 12$ Byte (x_i (double), i (long)) and 16 processors connected by a 16 Byte/packet bus. The matrix is 'bcspwr10' from the Harwell-Boeing collection (5300×5300, 8271 below-diagonal nonzeros; available in 'netlib'); other matrices from the 'bcspwr' set yield similar results, as do randomly generated matrices.

Spectral mapping achieves the best results, followed by greedy mapping which offers a fast alternative. DSC-spectral improves DSC performance but cannot compete with the min-cut methods. It is worth noting that for the 'bcspwr05' matrix (443×443, 590 below-diagonal nonzeros), 16 CPUs and an 'infinitely fast' network (bandwidth 10^{30} MB/s), spectral mapping achieves 95 % efficiency while DSC achieves about 51%. Even in this case where communication delays can be neglected, the min-cut algorithm generates better mappings.

Atmospheric Ozone Simulation. In our second application, speedup is bandwidth-limited due to large data items. Our topology consists of two B MB/s buses with 8 equal-speed processors at each, connected by a B MB/s link. This topology represents typical bottleneck situations — in Figure 1, these could occur if a computation is distributed between the nodes in the UltraSPARC cluster and the 16-node SGI Origin. Figure 4 gives results for this problem (with 32 atmospheric levels). In this simulation we used a network packet size of 256 byte; this size is representative for commodity workstation interconnects which are well-suited for GTCM.

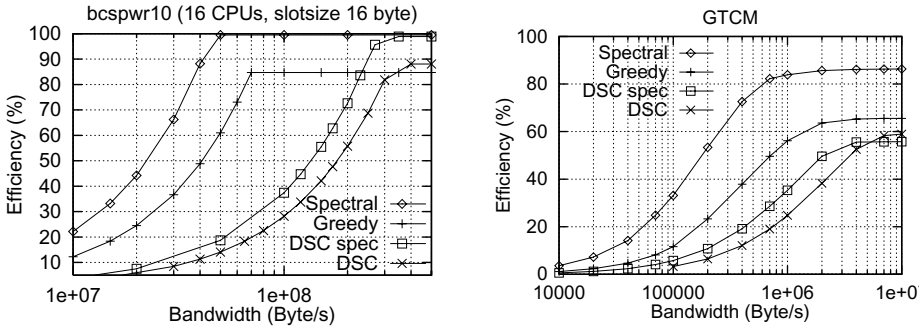

Fig. 4. Simulation results for STS (left) and GTCM (right)

Obviously, small cut size is essential: spectral mapping achieves the same efficiency as DSC for B a factor 10 smaller. Again, greedy mapping qualifies as a fast alternative with fair quality while DSC-spectral is slightly better than DSC without reaching the two min-cut methods.

5 Conclusions

The main contribution of our work is a paradigm for applying undirected-graph min-cut methods to dataflow graph min-cut mapping, by means of the 'time intervals' defined above. We have adapted spectral and greedy bisection to dataflow graph mapping. These methods are applicable to a wide range of processor/network speed ratios. We have demonstrated that, with respect to quality, min-cut mapping methods are slightly better than critical-path methods for fast networks where small cut size does not seem to matter, and that they are clearly superior for slow networks. Mapping speed is traded off for these advantages.

Future work includes the acceleration of spectral mapping in order to make it practical for a wider range of applications, and additional work on greedy spectral mapping in order to assess whether this method performs well for large dataflow graphs (greedy algorithms 'look' at the graph in a 'local' way) — toward this end, multilevel strategies as discussed in [16] for undirected graphs are promising. We have not considered strongly-heterogeneous clusters; for an application that consists, for example, of 'integer' as well as 'floating-point tasks' and runs on a mixed Intel and MIPS CPU cluster, this would lead to serious performance penalties. Finally, it is not clear whether our simplistic topology clustering method is appropriate for all network topologies appearing in practice.

If processor or network performances change during execution, it is necessary to compute a new mapping. For example, GTCM runs for several minutes to several days. During this time, network links and computers can break down and become available again; other users start and stop their own applications on a part of the clusters. It is impossible to adapt to these changes manually because 24-hour operator supervision would be necessary. The only alternative is to develop automatic mapping methods like the ones we describe. Greedy mapping is sufficiently fast for a remapping frequency

on the order of 1/second and is easily adapted to take initial data location into account. Spectral mapping is too slow except for long-running applications like our atmospheric simulation.

References

[1] S.T. Barnard. PMRSB: Parallel multilevel recursive spectral bisection. In *Proceedings of Supercomputing*, 1995.

[2] S.T. Barnard and H.D. Simon. A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems. In *Proc. 6th SIAM Conf. Par. Proc. Sci. Comp.*, pages 711–718, 1993.

[3] S.H. Bokhari. On the mapping problem. *IEEE Trans. Comput.*, C-30(3):207–214, 8/1981.

[4] S.W. Bollinger and S.F. Midkiff. Heuristic technique for processor and link assignment in multicomputers. *IEEE Trans. Comput.*, 40(3):325–333, March 1991.

[5] V. Chaudhary and J.K. Aggarwal. A generalized scheme for mapping parallel algorithms. *IEEE Transactions on Parallel and Distributed Systems*, 4(3):328–346, march 1993.

[6] F.T. Chong, S.D. Sharma, E.A. Brewer, and J. Saltz. Multiprocessor runtime support for fine-grained, irregular DAGs. *Parallel Processing Letters*, 5(4):671–683, December 1995.

[7] Fan R. K. Chung. *Spectral graph theory*. American Mathematical Society, 1997.

[8] G. Eisenhauer and K. Schwan. An object-based infrastructure for program monitoring and steering. In *Proc. 2nd SIGMETRICS Symp. Par. Dist. Tools*, pages 10–20, August 1998.

[9] V. Elling. A spectral method for mapping dataflow graphs. Master's thesis, Georgia Institute of Technology, 1998.

[10] V. Elling and K. Schwan. Min-cut methods for mapping dataflow graphs. Technical Report GIT-CC-99-05, Georgia Institute of Technology, 1999.

[11] U. Elsner. Graph partitioning — a survey. Technical Report SFB393/97-27, TU Chemnitz, SFB "Numerische Simulation auf massiv parallelen Rechnern", Dec. 1997.

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

[13] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[14] A. Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling DAGs on multiprocessors. *J. Par. Dist. Comp., Special Issue on scheduling and load balancing*, 16(4):276–291, 1992.

[15] W. Gu, G. Eisenhauer, K. Schwan, and J. Vetter. Falcon: On-line monitoring for steering parallel programs. *Concurrency: Practice and Experience*, 10(9):699–736, Aug. 1998.

[16] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. To appear in SIAM Journal on Scientific Computing.

[17] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell System Tech. J.*, 49:291–307, February 1970.

[18] T. Kindler, K. Schwan, D. Silva, M. Trauner, and F. Alyea. Parallelization of spectral models for atmospheric transport processes. *Concurrency: Practice and Experience*, 11/96.

[19] B. Plale, V. Elling, G. Eisenhauer, K. Schwan, D. King, and V. Martin. Realizing distributed computational laboratories. *Int. J. Par. Dist. Sys. Networks, to appear*.

[20] A. Pothen, H.D. Simon, and Kang-Pu Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11(3):430–452, July 1990.

[21] Tao Yang and A. Gerasoulis. List scheduling with and without communication. *Parallel Computing Journal*, 19:1321–1344, 1993.

[22] Tao Yang and A. Gerasoulis. DSC: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967, 1994.