75

A New Model for the Data Distribution Problem

Thomas Loos and Randall Bramley

1 Introduction

When solving a mesh-discretized PDE on a distributed memory parallel computer, two preliminary problems must be solved: the partitioning of the mesh and the mapping of partition sets to processors. These two define the data distribution problem. All partitioning algorithms try to minimize total computer solution time of the PDE, which is dominated by the execution time of the linear system solver on the resulting matrices. The algorithms attempt to minimize total solution time by approximately minimizing the load imbalance and communications overhead.

Current algorithms model the problem as one of partitioning the graph of the mesh. They estimate load balance in terms of equal size partition sets so that all sub-meshes have nearly the same number of nodes, and communications overhead is measured by the number of edges cut by the partitioning. These criteria are effective for simple iterative methods for solving linear systems, particularly methods based only on matrix-vector products. However, many problems of increasing interest in scientific computing generate linear systems that require preconditioners involving recurrences and other parallelism–inhibiting features.

Using the number of edges cut as a metric ignores the algorithm and data structures used for solving the linear systems. To better estimate the actual cost that partitionings induce on parallel iterative solvers, we define the approximate execution time (AET) of a linear system solver as the sum of its communications, memory, and computational times. The AET metric is calculated using a function that partially and inexpensively simulates the execution of a linear system solver on the target parallel computer.

For iterative linear system solvers the data to be divided are the structurally symmetric sparse matrix A, a sparse preconditioning matrix M, and vectors required to solve the system $Ax = b$. The data distribution problem is that of determining the data layout or distribution among the processors and is usually viewed as an instance of the graph partitioning problem [Saa96].

Graph partitioning algorithms try to minimize the number of edges cut in a graph G by a partitioning P subject to balance conditions. Graph-based metrics for quality

Ninth International Conference on Domain Decomposition Methods
Editor Petter E. Bjørstad, Magne S. Espedal and David E. Keyes

other than the number of edges cut have been proposed (see Ashcraft and Liu [AL95] and Rothberg [Rot96] for discussions and comparisons of these metrics). The typical iterative solver user views accuracy and execution time as the most important metrics. The iterative solver execution time ET can be calculated as $ET = \mathrm{NI} \times \mathrm{TPI}$, where NI is the number of solver iterations and TPI is the time per iteration. As a practical matter, NI cannot be determined ahead of time, so only the time per iteration can be estimated. Our results show the edges cut metric does not provide even a qualitative measure of the time per iteration.

A new metric, the approximate execution time (AET) metric, is proposed here to replace the edges cut metric as the cost function to be minimized by a graph partitioning algorithm. It is assumed that the solver algorithm can be stated as a sequence of calls to *kernel* functions. Then, the AET function estimates the time for each kernel function, accounting for the solver input data and computational environment. These kernel estimates are summed to give a time per iteration estimate for the solver; current estimates are within 10% relative error, but are generally much better.

2 The Data Distribution Problem

Classically, the structure of A is viewed as an adjacency matrix $\mathcal{A}$ for the data distribution problem. The data distribution problem is then solved by a graph partitioning algorithm, whose inputs are a graph G, specified in this case by $\mathcal{A}$, the number n of partition sets desired, and potentially an initial *partitioning* or division of G into n partition sets. The graph partitioning algorithm partitions $\mathcal{A}$ into n sets, each each corresponding to one solver process. After the partitioning algorithm is run, a *mapping* algorithm determines the logical process to physical processor mapping. Many graph partitioning algorithms assume G has undirected edges, so $\mathcal{A}$ and by extension, A, are assumed to be structurally symmetric. Frequently P is used to reorder and divide A into block rows or columns; in this paper, we assume that P partitions A into n sets that correspond to n block rows. Also, if process i is assigned a row r of A by P, it is assigned row r of M and of all vectors used by the solver.

Most existing partitioning algorithms minimize the number of edges cut of G by P. This has an apparent mapping to the solver execution time by assuming minimizing the number of edges cut corresponds to minimizing the amount of communication between processes. If it can be further assumed that solver execution time is dominated by communications time, the edges cut metric should predict iterative solver execution time. However, in practice communications time also depends on the actual number of messages since each incurs latency. Simply using the edges cut metric also does not account for the computer network topology, which can affect communications costs, or the solver, preconditioning algorithm, and speed of the parallel processing hardware, all of which affect the numerical costs.

Measuring Solver Execution Time

The iterative solver execution time is initialization time plus $\mathrm{NI} \times \mathrm{TPI}$, where NI is the number of solver iterations, and TPI is the time per iteration. Most of the

Table 1 Numbers of Bi-CGSTAB iterations required Laplace operator matrix, SHERMAN5, and BFS for eight different summation orderings of the dot products. Data from Etsuko Mizukami.

Ordering Method	1	2	3	4	5	6	7	8
Laplace	64	62	63	65	64	60	60	63
SHERMAN5	1034	1025	1024	998	908	974	870	934
BFS	46	49	49	38	37	49	49	38

initialization time is spent in computing a preconditioner, which is typically small compared to total solution time.

It is impossible in general to adequately estimate the number of iterations a preconditioned nonsymmetric iterative solver will take. Although *upper bounds* have been established for the number of conjugate gradient iterations needed for some simple problems with known eigenvalue distributions [AL86], no realistic estimates for practical problems are available. Three additional complicating factors also occur. First, the targeted systems are nonsymmetric. In this case, even a complete *a priori* knowledge of the eigenvalues of the system does not allow estimating the number of iterations. Secondly, all practical solvers use some form of preconditioning. Except in special cases such as diagonally dominant M-matrices, even the existence of the preconditioner is suspect, and its effect on the number of iterations not known. In many cases a preconditioner can actually increase the number of iterations required.

Finally, each domain decomposition implicitly defines a reordering of the matrix with subsequent changes in the order of operations and quality of preconditioning. Table 1 shows the number of iterations required by Bi-CGSTAB for the matrix SHERMAN5 from the Harwell-Boeing collection of test matrices, a steady-state backward-facing step problem in CFD, and the Laplace operator on a $24 \times 24 \times 24$ cube discretized using centered differences. Only the order of summation used in computing the dense dot products was varied; the partitioning and other computations were kept fixed. Even this simple change causes the number of iterations to vary by over 20%. Although this seems an unusual result which indicates an ill-conditioned system or unstable algorithm, it commonly occurs even with well-conditioned problems: the three in Table 1 have estimated condition numbers of 4.2×10^2, 3.6×10^5, and 1.3×10^4. CG-like iterative methods rarely have monotonic convergence with respect to the residual norm, and even CG applied to symmetric positive definite systems characteristically has sharp drops followed by plateaus. If the solver succeeds in reaching the termination residual norm right after a sharp drop, it can take many fewer iterations than if it is just above the termination point. Then the residual norm often stays above the termination level until the next sharp drop is encountered.

Other factors also contribute to the variability of numbers of iterations when the summation order changes. Most modern processors have combined multiply-add units, which send a full 106-bit mantissa from the multiply operation to the add unit. When the dotproduct is distributed across processors, however, the partial sums are rounded to 53-bit IEEE standard mantissas to be sent in a standard 64-bit word, changing the

Table 2 Partitioning Methods Studied

Method	Laplacian EC	BFS EC
Linear	21 364	9 276
Linear-KL	23 542	9 488
Multi-Level	8 792	6 702
Random-KL	22 839	10 674
Scattered-KL	34 532	9 482
Spectral	17 386	7 672
Spectral-KL	14 033	7 684

final sum. Another source of variability is the sensitivity of nonsymmetric iterative methods. Because they typically use an indefinite inner product, large oscillations can occur in the residual norm during the solve. Finally, nonsymmetric linear systems often have poor behaviour not predicted by the spectral condition number. Two well-known examples of this are a large departure from normality and having large magnitude eigenvalues lying close to the imaginary axis.

In a parallel environment, the summation ordering problem is further compounded by the unpredictable order of summation between the processors which affects the matrix-vector as well as the dot product operations. Since determining the number of iterations induced by a domain decomposition is impractical, the AET metric concentrates on the other factor of the total solution time: the time per iteration. This unpredictability of the number of iterations, however, is also unaddressed by the standard edges-cut metric which only targets the communication cost of a single iteration.

The Edges Cut Metric as a Time Per Iteration Predictor

To empirically test the edges cut metric as a predictor of the time per iteration, we used a parallel implementation of van der Vorst's bi-conjugate gradient (Bi-CGSTAB) algorithm [vdV92] with two block preconditioning algorithms: Jacobi/block diagonal (BDIAG) and block SSOR (BSSOR). The preconditioning matrix M consisted of the off-diagonal blocks of A and the factored diagonal blocks of A using incomplete LU factorization with 0 levels of fill (ILU(0)). Other iterative solvers and preconditioners are incorporated in the code, but this combination was chosen as typical of parallel nonsymmetric solvers and uses the kernels found in most parallel iterative methods. The solver was run on an Power Challenge with 2 GB of main memory using 8 of the 10 R-8000 CPUs. Two test matrices were used: the Laplacian operator using seven-point centered differences on a $50 \times 50 \times 60$ domain and *BFS*, a matrix of order 20 284 with 452 752 non-zeroes resulting from solving a refined backward facing step problem. Fig. 1 shows a view of the matrix generated by the Emily [BL94] matrix visualization tool. Seven octa-partitionings of the two matrices were generated using Chaco [HL93] using the methods listed in Table 2. In the table, "KL" means the local Kernighan-Lin [KL70, FM82] method was used as a post-processing step and "EC" is the number of edges cut. The timing results are shown in Fig. 2.

Figure 1 View of the BFS matrix with linear octa-partitioning. The matrix entries appear along the main diagonal. The horizontal and vertical lines represent the partitioning's division of the matrix into blocks.

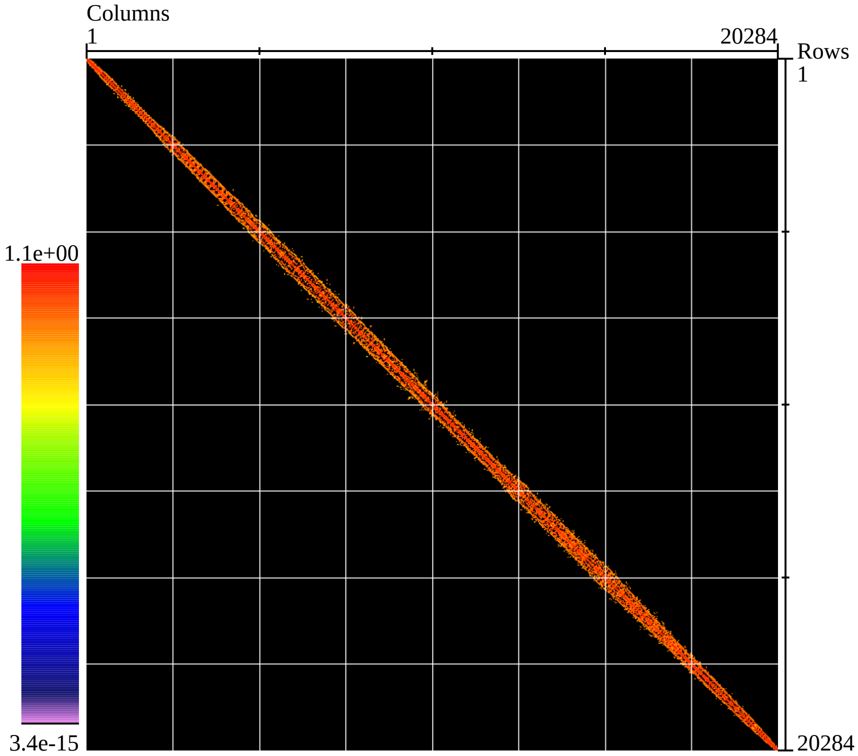
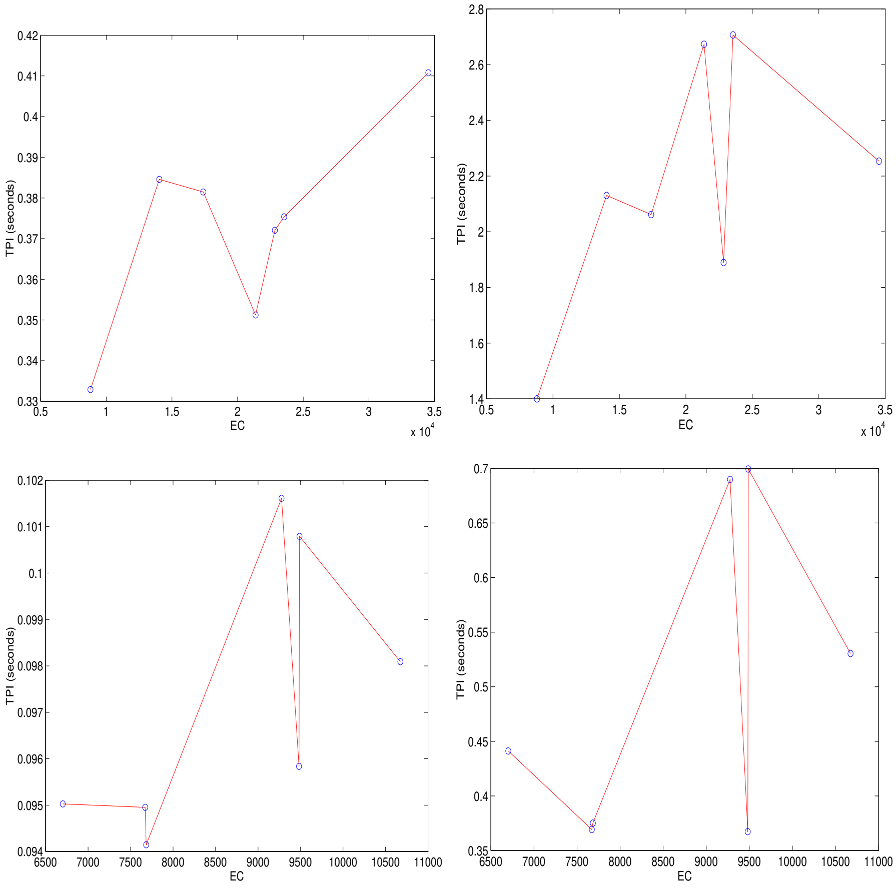


Figure 2 CGSTAB Time Per Iteration results. Top left: Laplacian with BDI AG. Top right: Laplacian with BSSOR. Bottom left: BFS with BDIAG. Bottom right: BFS with BSSOR.



The x-axis of each graph in Fig. 2 shows the number of edges cut for each partitioning method and the y-axis shows the observed time per iteration. If edges cut predicts the time per iteration, each graph in Fig. 2 should be monotone increasing. The simplest matrix/preconditioner pair is the regularly structured Laplacian matrix with the perfectly parallel BDIAG preconditioner. The results for this pair shown in the upper left graph of Fig. 2 indicate edges cut does predict the minimum and and maximum time per iteration correctly, but the function is clearly not monotone increasing. The upper right graph of Fig. 2 shows the results of using the same Laplacian matrix with the BSSOR preconditioner. The number of edges cut does not change, but the time per iteration function is clearly significantly affected, since nothing in the edges cut calculation accounts for a preconditioner change.

The lower left graph of Fig. 2 shows the results of the BFS matrix with BDIAG preconditioning, where the edges cut metric fails to predict the minimum *or* maximum time per iteration. The lower right graph of Fig. 2 shows the results of the BFS matrix using BSSOR preconditioning. The minimum number of edges cut for all partitionings of BFS is 6 702, yet the minimum time per iteration (0.367 s) occurs for the partitioning with 9 488 edges cut, closely followed by the partitionings at 7 672 (0.369 s) and 7 684 EC (0.375 s). The maximum time per iteration (0.699 s) occurs for the partitioning with 9 488 edges cut. The difference in edges cut between the partitionings with the minimum and maximum time per iteration is 6; yet the ratio of maximum to minimum time per iteration is 1.90. The edges cut metric does not even provide a qualitative prediction of the time per iteration.

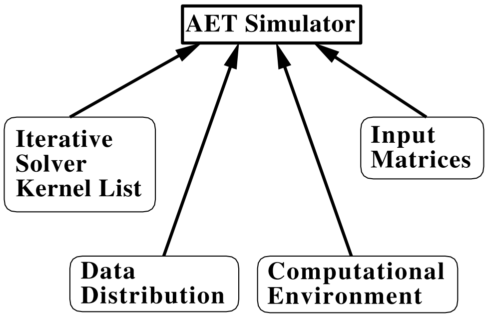
3 The AET Function

Fig. 3 outlines a basic software structure for the approximate execution time (AET) calculation. The AET function is input a list of high-level kernel operations representing the solver algorithm, A and M, low-level kernel timing data for the computational environment(s) used, the data distribution for A, and a representation of the processors in the computational environment. A "building block" approach generates the AET value by simulating a sequence of *kernel* calls.

We assume that a solver can be coded as a sequence of calls to a small number of *kernel* functions. This programming style allows for clear algorithmic statements [BBC$^+$94] and the use of standard numerical and communications library routines, such as the BLAS and the Message Passing Interface standards[MPIF94]. Solvers are expressed in terms of *high-level* kernels; high-level kernels operate on whole matrices and vectors. These high-level kernels are assumed to implemented in terms of either other high-level kernels or *low-level* kernels, which operate on vector and matrix blocks. For the AET calculation, each low-level numerical kernel is timed over a large range of inputs on one processor of the parallel system. Those observations provide a runtime estimation function for the kernel over inputs of arbitrary size. Point-to-point communication and synchronization kernels are timed in a parallel environment [LB96] to build similar models. The key parameters for these low level models for a particular parallel processor are stored in a data file. This allows the view of a parallel computer as a collection of kernel timing models.

The AET function simulates the execution of each high-level kernel in the iterative solver kernel list. Where the high-level kernel calls a low-level kernel, the AET function calculates the low-level kernel's input size and computational environment. From the parallel processor, kernel name, and input size a low-level kernel timing estimate is generated. The high-level kernel estimation routine then combines this estimate with information about cache and synchronization effects and sums the resulting estimate with previous estimates for that kernel to get a high-level kernel execution time estimate. Finally, these high-level kernel estimates are summed to get a time per iteration estimate for each physical processor in the computational environment. For the results below, these time per iteration estimates are averaged over all simulated processors.

Figure 3 AET Software Structure – the rectangles with rounded corners represent data objects. The model allows for changes in the solver, computational environment, and input matrix. This structure is used to generate a fairly accurate estimate of solver runtime.



Kernel Modeling

As an example of low-level kernel modeling, consider the dot product operation. The parallel `dotprod()` high-level kernel can be written in terms of the uniprocessor dot product `ddot()` and parallel reduction `reduce()` low-level kernels as follows:

```
double dotprod(Vector x, Vector y) {
   double sum, answer;
   sum = 0.0;
   for (each block b resident on this processor)
      sum = sum + ddot(x.blocks[b], y.blocks[b]);
   answer = reduce(sum);
   return(sum);  }
```

The AET function supports two low-level kernel modeling methods: a general piecewise linear model and a cached data model. The cached data model estimate is $E(N, S, t_{small}, t_{large}, t_{limit}) = N \cdot t_{est}(N, S, t_{small}, t_{large}, t_{limit})$, where N is the data size, S is the processor cache size S, t_{small} is the time per operation for cache resident data sets, t_{large} is the time per operation for non-cache resident data sets, and t_{limit} is modifies the estimate if the the caching strategy significantly alters the cost per operation. t_{est} is defined by:

$$t_{est} = \begin{cases} t_{small} & , \quad N \leq S \\ \min\{t_{limit}, (St_{small} + (N-S)t_{large})/N\}, & N > S. \end{cases}$$

This approximation for the dot product along with an estimate of the reduction cost [LB96] were summed and used to generate Fig. 4. The left hand graph of the figure shows the total time for dot product operation on one CPU of an SGI Power Challenge. The right hand side graph shows the dot product time per double, which outlines the effect of the t_{small}, t_{large}, and t_{limit} parameters.

Figure 4 Modeling of simple kernels. The left hand graph shows the accuracy of the model and the right hand graph shows the effect of the modeling parameters.

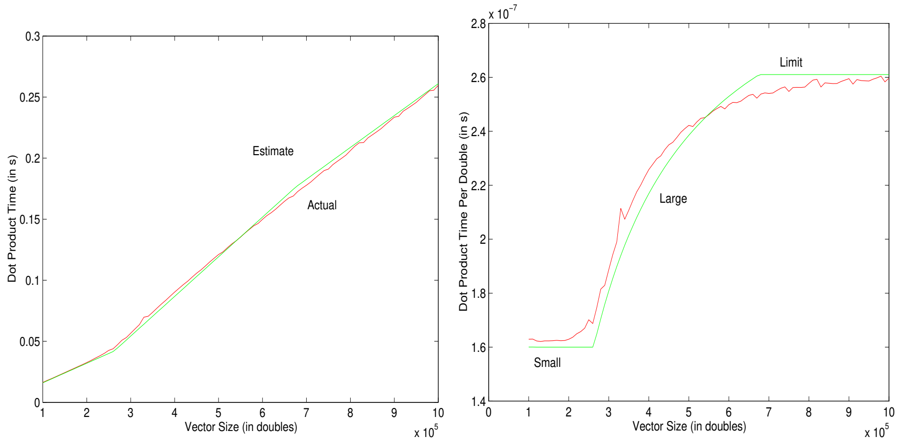


Table 3 AET Results for the Bi-CGSTAB solver using BDIAG preconditioning on the BFS and Lap150 matrices (Laplacian of order 150 000).

# CPUs	Lap150 Act TPI	Lap150 Est TPI	Lap150 Rel Err	BFS Act TPI	BFS Est TPI	BFS Rel Err
1	2.52	2.47	2.14 %	0.789	0.804	1.92 %
2	1.30	1.22	6.38 %	0.372	0.390	4.75 %
4	0.658	0.609	7.52 %	0.174	0.164	5.84 %
8	0.313	0.291	7.10 %	0.0910	0.0824	9.41 %

Complete Modeling

Stand-alone low-level kernel timings are not adequate for approximating the execution time because: (a) the size and distribution of the input matrices A and M are not known *a priori* and (b) residual effects such as the contents of the cache and synchronization delays from previous kernel calls are important. The previous cache contents can and do greatly change the cost of memory accesses [Sto90]; because of the change in cache hit ratios. For example, the vector copy low-level kernel has the ratio $t_{limit}/t_{small} = 8.1$. The AET simulator assumes the use of a least recently used cache replacement policy with a correction constant for other policies such as the random replacement policy used on the SGI Power Challenge. For simplicity, inter-CPU overhead is assumed to be mainly synchronization overhead.

Table 3 shows AET function results on the SGI Power Challenge on two test matrices: the Laplacian operator on a $50 \times 50 \times 60$ domain and *BFS* for the Bi-CGSTAB solver run on 1, 2, 4, and 8 SGI Power Challenge R8000 CPUs using BDIAG preconditioning. Both matrices were partitioned using a linear octa-partitioning. The AET calculation is accurate to within 10 %.

Related Work

Blau's [Bla92] work uses a run-time estimate as input to a partitioning algorithm used by a computer rendering system. This work used previous timing results to predict future timing results; a natural choice for a frame-by-frame renderer, where the input changes a small amount from frame to frame. It did not readily account for changes in the rendering algorithm or computational environment.

Adve [Adv93] and Xu, Zhang, and Sun [XZS96] also use a modeling strategy based on combining empirical observations. They identify segments of a program by first locating communication and synchronization points and computing a *task graph*. Each task then is timed – either in a uniprocessor environment on the same input data in Adve's system or on the same computational environment using a scaled-down version of the input in Xu, Zhang, and Sun's system – and the synchronization and communications routines are separately timed. The task graphs are used to drive a high-level simulator which accounts for inter-process memory contention, communication, and synchronization delays. Our system uses a similar general framework – low-level models based on direct timing observations are combined by a high-level model to get an estimate. As each kernel estimate is independent of the others, task graphs and task timing are not needed once the low-level kernel models are generated. This independence comes at a loss of generality – this model will not work for any but a subset of all parallel programs. However, a more accurate estimate is attainable by focusing on parallel iterative solvers, as this work shows.

4 Conclusions and Future Work

For the data distribution problem, solver run-time is the true metric for a partitioning. In practice the number of solver iterations cannot be predicted, and time per iteration is the primary factor in execution time that can be predicted for iterative linear solvers. We have developed an estimation function that reliably estimates the time per iteration and plan on using it as a cost function for partitioning algorithms. Further work includes using the AET to drive a partitioning algorithm, extending our system to more computational environments, and refining the high-level modeling scheme.

REFERENCES

[Adv93] Adve V. (October 1993) *Analyzing the Behavior and Performance of Parallel Programs*. PhD thesis, University of Wisconsin-Madison.

[AL86] Axelsson O. and Lindskog G. (1986) On the eigenvalue distribution of a class of preconditioning methods. *Numer. Math.* 48: 479–498.

[AL95] Ashrcraft C. and Liu J. (November 1995) Using Domain Decomposition to Find Graph Bisectors. Technical Report CS-95-08, York University.

[BBC+94] Barrett R., Berry M., Chan T., Demmel J., Donato J., Dongarra J., Eijkhout V., Pozo R., Romine C., and van der Vorst H. (1994) *Templates for the Solution of Linear Systems: Buildings Blocks for Iterative Methods*. SIAM, Philadelphia PA, first edition.

[BL94] Bramley R. and Loos T. (July 1994) EMILY: A Visualization Tool for Large Sparse Matrices. Technical Report TR 412A, Indiana University Computer Science

Department.

[Bla92] Blau R. (December 1992) *Performance Evaluation for Computer Image Synthesis Systems.* PhD thesis, University of California - Berkeley. Also available as UCB Techreport CSD-93-736.

[FM82] Fiduccia C. M. and Mattheyses R. M. (1982) A Linear Time Heuristic for Improving Network Partitions. In *Proceedings of the 19th ACM/IEEE Design Automation Conference*, pages 175–181.

[HL93] Hendrickson B. and Leland R. (October 1993) The Chaco User's Guide Version 1.0. Technical Report SAND 93-2339, Sandia National Laboratories.

[KL70] Kernighan B. W. and Lin S. (Feburary 1970) An Efficient Heuristic Procedure for Partitioning Graphs. *Bell Systems Technical Journal* 49: 291–307.

[LB96] Loos T. and Bramley R. (1996) MPI Performance on the SGI Power Challenge. In *Proceedings of the Second MPI Developer's Conference*, pages 203–206. IEEE Computer Society Technical Committee on Distributed Processing.

[MPIF94] Message Passing Interface Forum (May 1994) MPI: A Message-Passing Interface Standard. Technical Report UT-CS-94-230, University of Tennessee, Knoxville.

[Rot96] Rothberg E. (January 1996) Exploring the Tradeoff Between Imbalance and Separator Size in Nested Dissection Ordering. Technical Report none, Silicon Graphics, Inc.

[Saa96] Saad Y. (1996) *Iterative Methods for Sparse Linear Systems.* PWS Publishing Company, Boston, MA, first edition.

[Sto90] Stone H. S. (1990) *High Performance Computer Architecture.* Addison-Wesley, Reading, MA, second edition.

[vdV92] van der Vorst H. (1992) Bi–CGSTAB: A fast and smoothly converging variant of Bi–CG for the solution of nonsymmetric linear systems. *SIAM Journal of Scientific and Statistical Computing* 13: 631–644.

[XZS96] Xu Z., Zhang X., and Sun L. (1996) Semi-Emperical Multiprocessor Performance Predictions. Technical Report TR-96-05-01, University of Texas, San Antonio, High Performance Comp. and Software Lab.