A NEW MODEL FOR SOLVING THE DATA DISTRIBUTION PROBLEM

Thomas J. Loos

Submitted to the faculty of the University Graduate School
in partial fulfillment of the requirements
for the degree
Doctor of Philosophy
in the Department of Computer Science
Indiana University

December 1996

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Doctoral Committee

Randall Bramley, Ph.D.

Dennis Gannon

David Wise

Michael Jolly

November 15, 1996

Acknowledgements

There are a lot of people who've helped me along the long and winding path to a Ph.D. and I'd like to say thanks to some of them. I'm sure I've forgotten some helpful and friendly people, to whom I apologize in advance.

First, thanks to Randy Bramley, who got this thesis started, had lots of good ideas, took care of me in both the best and worst of times, and kicked me as hard as I needed, but no harder, along the way. I will always remember his constant admonition to "Do." Thanks to the rest of my committee – Dennis Gannon, David Wise, and Michael Jolly – for taking the time and trouble to help me correct and clarify this dissertation. Thanks to Pam Larson for sheparding me and my paper work through the University bureaucracy. She has been a real life saver, especially this last semester. This work has been supported by NSF Grants CDA 93-03189, CDA 93-09746, and ASC 95-02292.

I've met a lot of great people over the last five years in Lindley Hall. Marc Van

Heyningen was my first great compatriot, helping me out with class work and a great discussion partner for everything from geography to the future of the Internet. I cannot thank him enough for being in my corner at the debacle that was my oral exam. He's the first guy I could admit was smarter than me without grimacing inside. Phil Bradford showed me that you can play the Ph.D. game and still have fun with it. Emily Meyers taught me a lot of things that don't show up in this thesis. She's been a great source of motivation during the last couple of years and I wish her all the best. Charlie the night janitor has shared with me the great wisdom he's garnered over his years of experience – he knows more and has told me more about life than most of the academics I've met. I really enjoyed playing poker and swapping stories with the Poker Dogs – Bill Dueber, Byron Long, Vikram Subramaniam, Lowell Vaughn, and Dave Wilson. I'd especially like to thank Bill for sharing his insight with me when I needed it most.

The people in the Graphics Lab have been wonderful. Pete Shirley let me in the door and got me started toward a Ph.D. He taught me to work at least one hour per day, which has been wonderful. Paulo Maciel and Shankar Swamy inspired me with their high personal ethics. Eric Wernert is an inspiring teacher and wonderful storyteller. Kurt Zimmerman has been the guy I talk to about the motivations for getting a degree and has introduced me to some great music. Along with Kurt, Jason Almeter, Rajesh Kamath, and Manish Khettry helped me study for the screaming

exam. I had a great time studying with these terrifically smart and interesting people, even if the test was a nightmare. Alan Keahey has given me some of the best pieces of advice in my life – he is a wonderfully calm and insightful person to talk with, especially over a beer or two. There have been two especially great friends in the Graphics Lab day-in and day-out during my last year. Chun-Perng Cheah has been a great guy to talk with and he wrote the first version of what became my thesis solver. Deepa Viswananthan has listened to me ramble while I sorted through the things during the end game. Her friendship, courage, and unflagging good humor have been a great inspiration to me.

Finally, a great big "Thank You" to my family, Mom, Dad, Jerry, Kelly, and Heidi, who encouraged me to become educated beyond my intelligence. I have been truly blessed by being part of a wonderful family who have always supported each other.

Abstract

Thomas J. Loos

A NEW MODEL FOR SOLVING THE DATA DISTRIBUTION PROBLEM

Solving large linear systems on parallel computers requires first solving the data distribution problem. For a sparse coefficient matrix A, that problem typically is to assign rows of A to processors, and is usually solved by partitioning the graph of the matrix. Since the graph partitioning problem is NP-hard [11], practical algorithms are based on heuristic approaches. Performance of graph partitioning algorithms is measured in terms of the number of edges cut (EC): the lower the number of EC, the better the algorithm. For the data distribution problem, an additional load-balancing constraint is also placed on the algorithm.

Application scientists are interested in another metric: the time it takes to solve the system. Comparing several different partitioning algorithms on matrices derived

from irregular grid PDE problems, results indicate that the number of EC is a poor predictor of solver run-time. This is because current graph partitioning algorithms are not tailored for the data distribution problem; the linear system solver, preconditioner, matrix data structures, and computational environment are generally ignored.

A simple simulation-based approximate execution time (AET) metric that estimates the performance of a linear system solver based on combining cost estimates of the solver's computational, communications, and synchronization kernels is presented. Simulation of kernel performance provides a comparatively easy way to model solvers and preconditioners, while giving reasonably accurate estimates in a reasonable amount of simulator execution time.

Results are presented comparing actual time per iteration values with AET estimates on the SGI Power Challenge and IBM SP-2 parallel computers. AET estimates have a relative error of 13.1% or less compared to the actual time per iteration of the HMPE linear system solver. Typically, the AET estimate is within 5% of the actual TPI. A data distribution algorithm is proposed using the AET metric as its cost function.

Contents

List of Tables

List of Figures

1

Introduction

Solving linear systems of equations is a common problem in scientific computing. Frequently, they arise from the discretization of partial differential equations (PDEs) over some domain. A common method for the numerical solution of PDEs is the finite element (FE) method. The FE method divides the domain, usually a region in $\Re^2$ or $\Re^3$, into a mesh of **elements**. Each element is a simple polygon or solid, such as a quadrilateral or tetrahedron. A mesh need not be made of of only one type of element, and the key importance of the FE method is its ability to model complicated geometries. Other discretization strategies, such as finite differences and finite volumes, also lead to a mesh.

A parallel processor is often used for solving PDEs, both for the speed provided by many processors working on the same problem and increasingly, because the problem is so large that it cannot fit on one processor. To solve a PDE on a parallel processor

two additional problems need to be solved: the **partitioning** of the mesh into smaller meshes for each processor to compute and the **mapping** of the partitioned meshes to processors. To solve the PDEs, the grid points of the discretized meshes are assembled into matrix form and the resulting linear system is solved, commonly using an iterative linear system solver. The **data distribution** problem is the problem of determining the data layout or distribution among the processors so that the iterative linear system solver execution time is minimized – see Figure 1.1.

The communications overhead required for a solving the systems that arise from the partitioning of a mesh is related to the number of edges cut during the partitioning of the mesh, so the number of edges cut is generally used to measure the quality of the partitioning. Graph-based metrics for quality other than the number of edges cut have been proposed (see Ashcroft and Liu [13] and Rothberg [23] for discussions and comparisons of these metrics.) During the mapping phase, the number of edges cut can also be weighted to allow for variable interprocessor communications costs caused by interprocessor connection topology.

Current graph partitioning algorithms concentrate on one or two important pieces of the solution process – the data set of interest and the underlying hardware interconnection topology. This metric ignores other important pieces of the solution process: the algorithm and data structures used for solving the system. Not all algorithms use the same memory access patterns, or have the same memory requirements, or

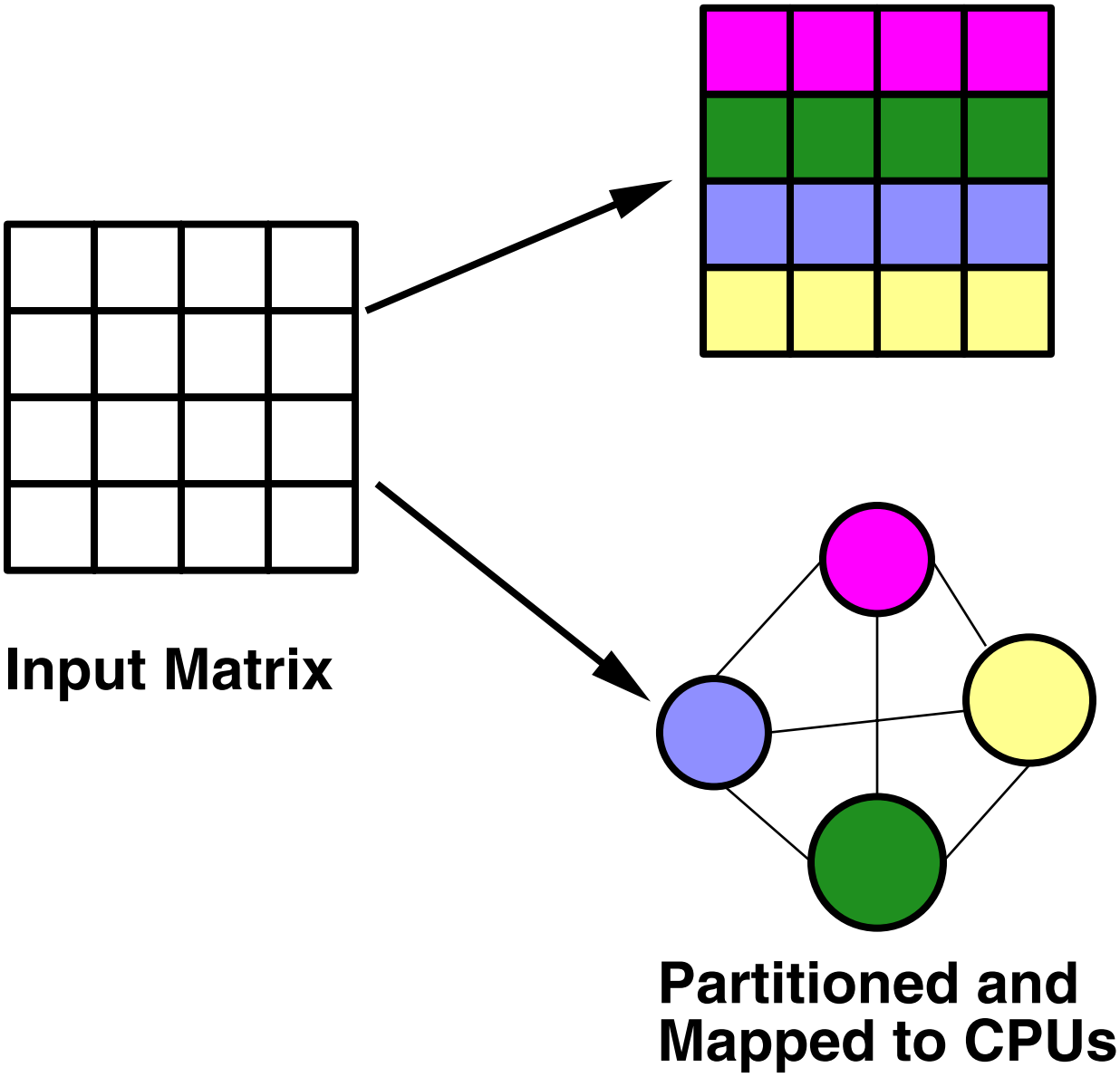

Figure 1.1: The **data distribution** problem is the problem of determining the data layout or distribution among the processors so the iterative linear system solver execution time is minimized.

even have the same order of execution time. A better approximation would take into account the communication required by the solution algorithm using the input data on the target computer, along with an approximation of the computational costs for each processor in a multiprocessor network, leading to the **approximate execution time (AET)** metric. Calculating the AET metric can be done in a function that cheaply simulates the execution of the algorithm on the target parallel processor. We propose replacing the edges cut metric with the AET metric in a data distribution algorithm. Since the AET metric accounts for the computational environment, the need for mapping algorithms would be eliminated.

1.1 Notation Used in the Thesis

Tables 1.1, 1.2, and 1.3 contain the notation used in this thesis broken down by subject.

1.2 Contributions of this Thesis

The primary contribution of this thesis is the definition and use of the AET metric to replace the edges cut metric as the measure of data distribution quality. The AET is calculated by an estimation routine (the AET simulator) which is an implementation of a model of parallel program performance that accounts for a program's code,

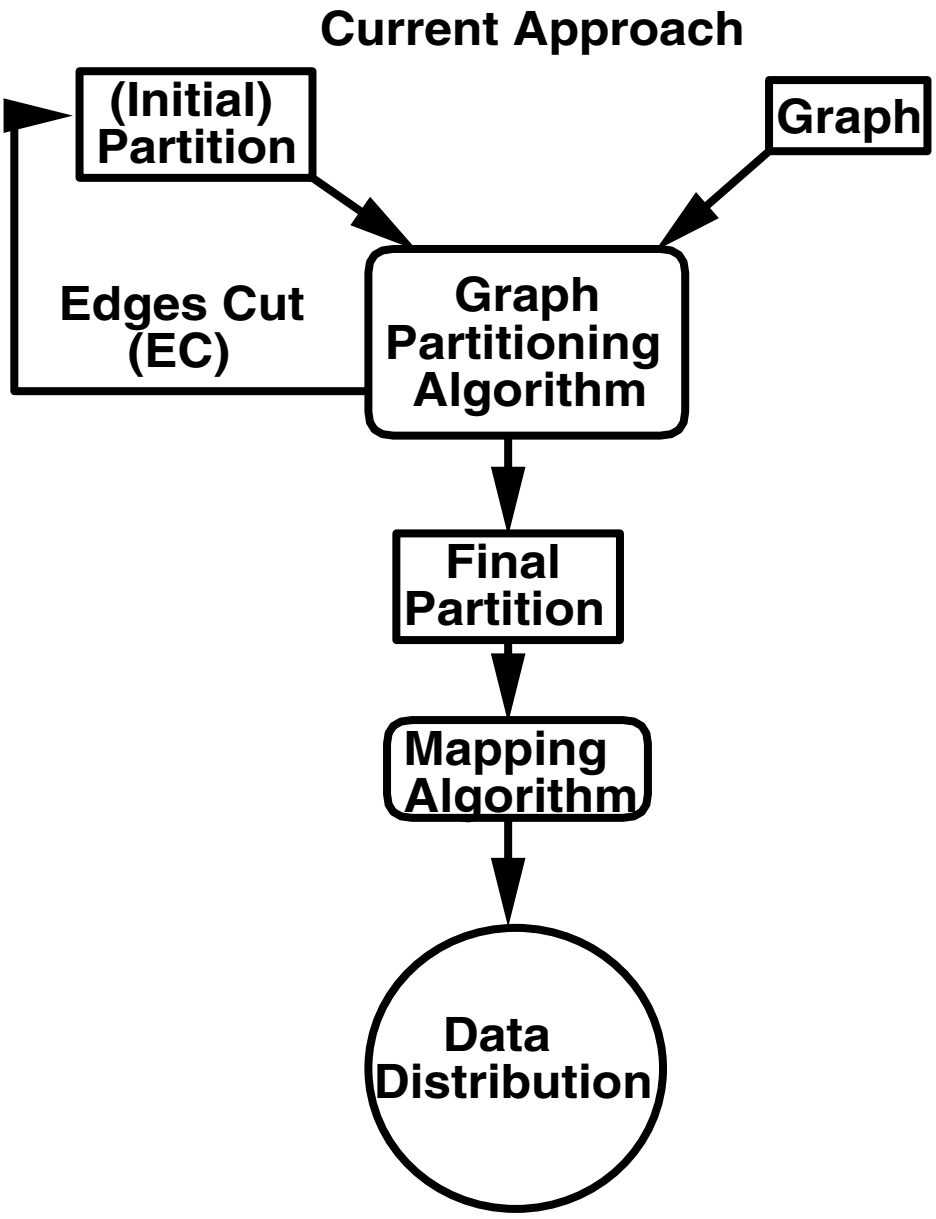

Figure 1.2: The current data distribution solution method is to use a graph partitioning algorithm to divide the data, then apply a mapping algorithm to map the partition sets onto processors. Graph partitioning algorithms attempt to minimize the number of edges cut.

Table 1.1: Graph Partitioning Notation Used in the Thesis

Name(s)	Description
E	Set of edges of G
EC	Number of Edges Cut
J	Number of partitions
G	Graph
P	Partitioning of G
V	Set of vertices in G
X	SPECT solution vector
$\mathcal{A}$	Adjacency matrix
$\mathcal{D}(v)$	Degree of vertex v
$\mathcal{L}$	Laplacian matrix of G
v	A vertex in G
e	An edge of G
c_i	Cut edge set for P_i
λ	Lagrange multiplier

Table 1.2: Modeling and Estimation Notation Used in the Thesis

Name(s)	Description
C	Parallel computer
H	Hardware type
K	Number of kernels
$\mathcal{M}$	A parallel machine
N	Number of processors
$\mathcal{N}$	Number of architectures in $\mathcal{M}$
Δ	A parallel program
T	Topology Matrix for C
$\mathcal{E}$	Estimated time
$\mathcal{P}$	Number of processes
p	A process
t	Time (generally subscripted)
$\mathcal{H}$	Hit ratio

Table 1.3: Linear System Notation Used in the Thesis

Name(s)	Description
$Ax = b$	Linear system to be solved
D	A diagonal matrix
L	A lower triangular matrix
U	An upper triangular matrix
$\mathcal{S}$	A Linear System Solver
NI	Number of Iterations
TPI	Time Per Iteration
NBR	Number of block rows
NNZ	Number of non-zero entries
R	Relative error
S	GMRES Krylov subspace size
SIT	Solver Initialization Time
Y	GMRES Krylov subspace basis
H	GMRES Upper Hessenberg matrix
a_{ij}	Entry of A
h_{ij}	An entry of H
d, w, y, z	Vectors
i, j, k, l, m	Integer variables
n	Order of matrix A
r	Residual vector
s	Maximum fill level in ILU(s)
α	Search vector update value
β	Search direction update value
γ, ζ, η	Constants
ϵ	Solver residual tolerance
ψ, ϕ	Polynomials formed by CGSTAB
ω_h	CGSTAB parameter

computational environment, and inputs without requiring highly detailed hardware specifications or assembly-level run-time traces. This parallel program performance model assumes that a parallel program is comprised of a combination of a known list of **kernels**, whose performance characteristics typify the performance of the parallel program. A kernel is any function that the performs a basic operation. The AET simulator estimates the kernel performance in "building block" fashion by estimating the performance of each kernel and summing those estimates.

The second contribution of this thesis is HMPE (HyperMatrix Parallel with Estimation), a parallel, preconditioned, iterative linear system solver. HMPE is a practical test of the ideas behind both the parallel program performance model and the AET calculation. It implements the two most robust nonsymmetric linear system solvers, two block preconditioners, and five factorization methods for local preconditioning. HMPE results indicate the number of solver iterations cannot be estimated and that the edges cut metric is a poor predictor of iterative solver performance, in terms of time per solver iteration. HMPE results also describe some requirements of any good solver estimation routine.

The third contribution of this thesis is the combination of the first two contributions: HMPE's ability to predict its own execution time. HMPE's implementation of the AET simulator generates qualitatively accurate run-time estimates of its performance, in terms of time per iteration.

1.3 Data Distribution Problem Statement

To execute a program on a parallel computer, the program's data and computation must be divided among the processors. Some parallel programming paradigms implicitly divide data by dividing computation, but this dissertation will concentrate on the explicit division of data between processors. In the specific case of an iterative linear system solver applied to discretized partial differential equations, the data to be explicitly divided are the structurally symmetric sparse matrix A, a sparse preconditioning matrix M, and vectors required to solve the system $Ax = b$. In this case, the **data distribution** problem is the problem of determining the data layout or distribution among the processors so that the iterative linear system solver execution time is minimized. It is usually viewed as a specific instance of the **graph partitioning** problem [50].

The data distribution problem is to minimize the amount of time solving a sparse linear system on a parallel computer given the following:

1. A (large) sparse linear system $Ax = b$, with a $n \times n$ matrix A and a $n \times 1$ vector b given. An example of such a system comes from the solution of a PDE using the FE method. n is commonly called the **order** of the matrix and can also be written as $|A|$.

2. A sparse linear system solution algorithm $\mathcal{S}$.

3. A parallel computer C running of $\mathcal{P}$ processes on N processors, with a connection topology $p \times p$ matrix T, where entry T_{ij} is 1 if processors i and j are directly connected in C and 0 otherwise. C can also be thought of as a processor graph. Assume each processor in C is running at most one process, so $\mathcal{P} \leq N$ T is a simplified version of the processor graph discussed in [26]. The communications cost t_{comm} between directly connected processors is assumed initially to be linearly related to the number of values (nvals) transmitted; that is,

$$t_{\text{comm}} = t_{\text{overhead}} + \text{nvals} \times t_{\text{gap}} \tag{1.1}$$

where t_{overhead} is the cost to initiate communication between processors, including message buffer loading and unloading, and t_{gap} is the average cost to transmit one value[16].

The problem is to execute $\mathcal{S}$ using C to solve $Ax = b$ while minimizing:

$$t_{\text{exec}} = \max_{\text{i=1...p}} \{t_{\text{calc}}(\text{i}) + t_{\text{comm}}(\text{i})\} \tag{1.2}$$

where t_{calc}(i) is the total time spent by processor i in calculations and t_{comm}(i) is the total time spent by processor i in communication.

This thesis is not primarily concerned with solving linear systems, but in modeling linear system systems with an eye toward cost-effective methods of partitioning data and performance prediction.

A fair amount of research has been put into the general problem of partitioning data, assuming the data are representable as a graph. The resulting problem is called the **graph partitioning** problem, which is important in a number of fields [28], including VLSI layout, run-time scheduling for parallel processing, and, as indicated, the solution of linear systems on parallel computers. Since it is an NP-hard problem [11], many heuristic approaches to the general graph partitioning problem have been proposed. After a data partitioning is generated, the problem of **mapping** the partitions onto processors must be solved. The mapping problem, which corresponds to the graph embedding problem, is NP-complete [26, p. 2], so heuristic approaches are also used to solve the mapping problem.

1.4 Summary of Graph Partitioning Algorithms

The current solution to the data distribution problem method is to use a graph partitioning algorithm to divide the data, then apply a mapping algorithm to map the partition sets onto processors. Graph partitioning algorithms as applied to the data distribution problem commonly attempt to minimize the number of edges cut. Six

graph partitioning methods are presented below which were used for thesis results.

1.4.1 Graph Partitioning Definitions

The following definitions are taken from those given in [33, 31], and primarily from [26, pp. 1-39].

1.4.1.1 General Graph Theoretical Definitions. A graph G is defined as $G = (V, E)$, where V is a set of vertices, $\mathcal{V} = |V|$, and $E \subset V \times V$ is the set of edges with $v_1, v_2 \in E$. A vertex v is a point in either $\Re^2$ or $\Re^3$. Unless otherwise stated, assume $v \in \Re^3$ and all edges are undirected; that is, (v_1, v_2) and (v_2, v_1) are the same edge. The degree $\mathcal{D}(v)$ of a vertex v is the number of edges in E that have v as an endpoint. For each edge $(u, v) \in E$, let there be associated a value $ew(u, v)$ known as its **edge weight** and, with each vertex $v \in V$ let there be associated a value $vw(v)$, called its **vertex weight**. Then G, together with the edge and vertex weights, is a **weighted graph**.

The adjacency matrix $\mathcal{A}(G) = [\mathcal{A}_{ij}]$ of an irreflexive graph G is defined as:

$$\mathcal{A}_{ij} = \begin{cases} 1 & : \quad i \neq j, (i, j) \in E \\ 0 & : \quad \text{otherwise} \end{cases} \tag{1.3}$$

The Laplacian matrix $\mathcal{L}(G) = [\mathcal{L}_{ij}]$ of a graph $G = (V, E)$ is defined as

$$L_{ij} = \begin{cases} -1 & : \quad i \neq j, (i,j) \in E \\ \mathcal{D}(i) & : \quad i = j \\ 0 & : \quad \text{otherwise} \end{cases} \tag{1.4}$$

1.4.1.2 Graph Partitioning Definitions. A **J-way partitioning** P of a graph $G = (V, E)$ is a set $P = \{P_1, P_2, \ldots P_J\}$ of nonempty, pairwise disjoint subsets of V such that $\cup_{i=1}^{J} P_i = V$. Each of the P_i's, $i = 1 \ldots J$ is a **partition**. Note that for the data distribution problem, typically $J = p$. J is assumed to be a power of 2. Corrections are available when J is not a power of 2.

A **cut-edge** e of a $J-$way partitioning P is an edge (v_1, v_2) such that $v_1 \in P_i, v_2 \in P_j$, with $i \neq j$ – it is an edge that connects vertices in different partitions. The **cut-edge set** $c_i, i = 1, 2, \ldots, j$ of a partition P_i is the set of cut edges with one vertex in V_i; for each partitioning $P = \{P_1, P_2, \ldots P_J\}$, there exist a corresponding set of cut-edge sets $\{E_1, E_2, \ldots E_J\}$. Define the cut-edge set $c(G)$ for the graph G to be $c(G) = \cup_{i=1}^{J} c_i$. Since all edges are assumed to be undirected, each cut edge will be in two cut-edge sets. In particular, for a 2-way partitioning or **bipartitioning**, $c(G) = c_1 = c_2$.

The **partition graph** $G_P = (V_P, E_P)$ of a J-way partitioning $P = \{P_1, P_2, \ldots, P_J\}$ is defined as: Let each vertex $V_{P_i} \in V_P, i = 1, 2, \ldots, J$ directly correspond to partition

P_i and let there be an edge $(i,j) \in E_P$ if and only if there is a cut-edge from P_i to P_j.

Let the edge weight $ew_P(i,j)$ of an edge $e = (i,j)$ be the cardinality of the cut-edges between P_i and P_j. Let the vertex weight $v(i)$ be any fixed estimate of the computational cost required by the partition P_i. Such estimates are often available for model problems, but are difficult to obtain in practice.

Unless otherwise stated, the partitioning methods below are assumed to solve one instance of the graph bipartitioning problem: create a bipartitioning $P = (P_1, P_2)$ of G such that the cardinality of the cut-edge set is minimized subject to the constraint that $|P_1| = |P_2|$, if $\mathcal{V}$ is even. If $\mathcal{V}$ is odd, the constraint is $||P_1| - |P_2|| = 1$. Assume below for simplicity that $\mathcal{V}$ is even. If more partitions are needed, the bipartitioning method can be recursively called until enough partitions are created. See Figure 1.3 for a diagram of the partitioning problem. Note that a partitioning method can be thought of as generating either a bipartitioning or a partition graph.

1.4.2 The Kernighan-Lin (KL) Partitioning Algorithm

The key idea of the Kernighan-Lin [41] algorithm (KL) is to look at the **gain** in moving a vertex from one partition to the other. To explain gain, assume that $v \in V$ and P is a partition and that there exists a cost function $\text{cost}(v, P)$ that returns the cost of $P \cup v$, if $v \notin P$, or the cost of P, if $v \in P$.

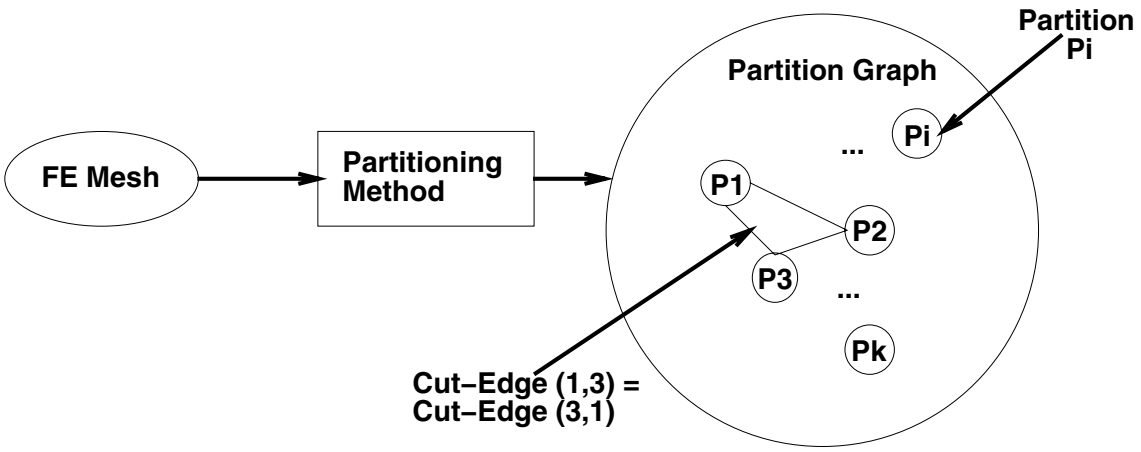

Figure 1.3: The Graph Partitioning Problem – the partitioning method can be thought of as generating either the partitions or the partition graph.

Assume without loss of generality that $v \in P_1$. The gain of a vertex is: $\text{gain}(v) = \text{cost}(v, P_2) - \text{cost}(v, P_1)$ or the cost of the new partition if v is moved to the other partition minus the cost of keeping the vertex in the same partition. The gain can be positive, negative, or zero. The gains will all certainly be negative if $P_1 = V$ and $P_2 = \emptyset$. The algorithm proceeds by swapping pairs of elements with the best net change in gain until either the total gain of the system is 0 or a predefined maximum number of iterations has been reached.

The original KL algorithm has $\mathcal{O}(\mathcal{V}^2 \log \mathcal{V})$ runtime [41]. Versions of KL with $\mathcal{O}(\mathcal{V})$ runtime have been found by both Fiduccia and Mattheyes [25] and Shiraishi and Hirose [53] – both algorithms achieved the linear run time by use of clever data structures to keep careful track of the gain values and the vertices whose gain values needed updating.

This is the most commonly used **local** graph partitioning algorithm; that is, it looks at information from only a small subset of the vertices at a time, not at the whole graph, which **global** methods do. KL's performance is heavily dependent on the given initial partitioning. It is frequently used to help "clean up" global methods [33, 29]. Other improvements and modifications have been made to KL which have made it the "recognized champion among classical approaches to the graph bisection problem" [10].

1.4.3 Linear (LIN) and Scattered (SCAT) Graph Partitioning Methods

Both the LIN and SCAT methods are based on simple assignment based on the numerical ordering of the nodes. Assume each vertex $v \in V$ is assigned an index $i, i = 1, 2, \ldots \mathcal{V}$, let v_i be the vertex whose index is i, and let J be the desired number of partition sets. Assume J evenly divides $\mathcal{V}$ and let $j = \mathcal{V}/J$ be the number of nodes in each partition set. The LIN method assigns vertices in block-wise fashion; vertices $v_{(l-1)j+1}, v_{(l-1)j+2}, \ldots v_{lj}$ are assigned to partition set $P_l, l = 1 \ldots J$. The SCAT method assigns vertices in cyclic fashion; vertices $v_l, v_{K+l}, \ldots v_{(j-1)K+l}$ are assigned to partition set $P_l, l = 1 \ldots J$. If J does not evenly divide $\mathcal{V}$, use some method (i.e. block-wise, cyclic, or even random) to assign the remaining vertices to partition sets.

The performance of both methods is dependent on the assignment of indices to vertices, but LIN frequently has a small to medium number of edges cut, and SCAT generally has a large number of edges cut.

1.4.4 Recursive Spectral Partitioning (SPECT)

The derivation of the SPECT method is described below and a sketch of the algorithm is provided. This description is paraphrased from [33, pp. 817-821].

Let the mesh to be partitioned be described by $G = (V, E)$. Assume the $\mathcal{V}$ vertices in the mesh are numbered 1 through $\mathcal{V}$. Each vertex i is assigned a value X_i of either $+1$ or -1 subject to a balancing constraint that is described below. Assign all vertices for which $X_i = 1$ into partition $P1$ and for which $X_i = -1$ into another partition $P2$. If an edge (v_1, v_2) is not cut by this partition, then $X_{v_1} - X_{v_2} = 0$. If (v_1, v_2) is cut, then $X_{v_1} - X_{v_2} = 2$ or -2, so $(X_{v_1} - X_{v_2})^2 = 4$. Then EC, the total number of edges cut, for the bisection of the mesh is:

$$\text{EC} = \frac{1}{4} \sum_{(v_1,v_2)\in E, v_1 > v_2} (X_{v_1} - X_{v_2})^2 \tag{1.5}$$

For load balancing, the number of vertices in each partition has to be equal. Since

by assumption $\mathcal{V}$ is even, the balancing condition becomes:

$$\sum_{i=1}^{\mathcal{V}} X_i \quad = \quad 0 \tag{1.6}$$

The partitioning problem then becomes minimizing (1.5) subject to (1.6). If we relax the problem and use its continuous formulation, the solution to the corresponding quadratic would be minimized if all the X_i's were 0. Add the constraint

$$\sum_{i=1}^{\mathcal{V}} {X_i}^2 \quad = \quad \mathcal{V} \tag{1.7}$$

to avoid this trivial solution, since each ${X_i}^2 = 1$. Let $\mathcal{L} = \mathcal{L}(G)$ be the Laplacian matrix of the mesh. Note that $\mathcal{L}$ satisfies

$$\mathcal{L} \times \mathbf{1} \quad = \quad \mathbf{1}^T \times \mathcal{L} = \mathbf{0} \tag{1.8}$$

where $\mathbf{0}$ and $\mathbf{1}$ are the $\mathcal{V} \times 1$ vectors of zeros and ones, respectively. Equation 1.5 can be expressed as a quadratic in X using $\mathcal{L}$:

$$EC \quad = \quad \frac{1}{4} X^T \mathcal{L} X \tag{1.9}$$

Using the method of Lagrange multipliers on the minimization of Equation 1.9

subject to Equations 1.6 and 1.7, we find that, for constants ζ_1 and ζ_2 :

$$\nabla(\frac{1}{4}X^T\mathcal{L}X) \;=\; \zeta_1\nabla(X^TX) + \zeta_2\nabla(X^T\mathbf{1})$$

or equivalently

$$\mathcal{L}X \;=\; \lambda_1 X + \lambda_2\mathbf{1} \tag{1.10}$$

where the λ's above are the Lagrange multipliers.

Pre-multiplying Equation 1.10 by $\mathbf{1}^T$ we get

$$\begin{aligned}
\mathbf{1}^T\mathcal{L}X &= \mathbf{1}^T\lambda_1 X + \mathbf{1}^T\lambda_2\mathbf{1} \\
\mathbf{0} &= \mathbf{1}^T\lambda_1 X + \mathbf{1}^T\lambda_2\mathbf{1} \;\text{ (from Equation 1.8)} \\
\mathbf{0} &= \lambda_1\mathbf{0} + \lambda_2 n \qquad (1.11)
\end{aligned}$$

Equation 1.11 implies $\lambda_2 = 0$, then by Equation 1.10 $\lambda = \lambda_1$ is an eigenvalue of $\mathcal{L}$. By Equation 1.9, EC $= \frac{1}{4}n\lambda$. To minimize EC, X has to be an eigenvector of $\mathcal{L}$ corresponding to the smallest eigenvalue that satisfies Equations 1.6 and 1.7. $\mathcal{L}$ is a positive semi-definite matrix with the smallest eigenvalue equal to 0 and corresponding eigenvector $\mathbf{1}$. Clearly, $\mathbf{1}$ does not satisfy the constraint $\sum_{i=1}^{n} X = 0$. To find the optimal value of X, set λ equal to the second smallest eigenvalue of $\mathcal{L}$ and set X to

the eigenvector corresponding to λ.

The trick is to use this real-valued vector X to partition the graph, since X will lead to an edge separator of the graph, by the following technique. Let m be the median value of X, let P^1 be the set of vertices for which $X(i)_{i \in V} \leq m$ and let $P^2 = V - P^1$. If $|P^1| - |P^2| > 1$, arbitrarily move enough vertices with X components equal to m from P^1 to P^2. Then, let c_{cut} be the set of edges that are cut by this partitioning; that is, those edges with one endpoint in P^1 and the other in P^2. Let P^1_{cut} be the vertices in P^1 that are endpoints of edges in c_{cut} and let P^2_{cut} be the vertices in P^2 that are endpoints of edges in c_{cut}. The graph $G_{\text{cut}} = ([P^1_{\text{cut}} \cup P^2_{\text{cut}}], c_{\text{cut}})$ is bipartite and Pothen et. al. [48] use an algorithm for calculating the minimum cover of the bipartite graph to find an approximate minimum edge cut. Hendrickson and Leland [29] use a multilevel approach (described below) to improve the quality of partitioning generated by the spectral method.

This method has been found [33, 30, 26, 60] to give good partitionings; in fact, SPECT gives the lowest number of edges cut partitionings of any purely global method. However, the cost of finding an eigenvector of a large sparse matrix can be high.

1.4.5 Simulated Annealing (SA)

SA is a local partitioning method, based on a general procedure of trying to find a global optimum by allowing occasional "bad" moves to ensure that the algorithm has not converged to a local minimum. SA is "motivated by an analogy to the behavior of physical systems in the presence of a heat bath" [37, p. 865]. An important concept in SA is that of temperature – when the temperature is high, the solution is allowed more flexibility to make locally bad moves, but as the SA algorithm runs, the temperature is cooled according to a "temperature schedule" until the solution is "frozen" to a (hopefully) optimum value. Each temperature value is used for a fixed number of iterations (the temperature's "length"). In this problem, SA's cost function is a partition's number of edges cut.

SA can be shown to converge to an optimal solution with probability approaching 1. [37, p. 868]. To practically test this claim, several authors have compared SA with KL and other algorithms [37] [38] [10] [26] and have generally found that SA gives reasonably good partitionings in practice but is sensitive to the values of the initial temperature and control values of the temperature schedule, and has considerably longer run times than the other algorithms.

SA has also been used to solve the mapping problem [60] [17]. The mapping results are similar to when SA is applied to the partitioning problem: the mappings are good,

but at the expense of long runtimes and difficulty in determining the appropriate control values.

1.4.6 Multilevel Partitioning Algorithms (MLP)

The basic idea of MLP is to merge a good global method with a good local method. This suggests using a global method to obtain an initial partitioning and then use a local method to refine the partitioning. Also, the graph itself can be simplified or **coarsened** to allow for faster partitioning [40, 4, 29], which can greatly improve the speed of a SPECT algorithm on a large mesh.

The general MLP algorithm is:

```
1. Refine/coarsen graph if desired.
2. while (not done) do
3.    Let P = global_partitioning_algorithm(G)
4.    Let P' = local_partitioning_cleanup_algorithm(G,P)
5.    done = finished(P')
6.    Further refine/coarsen graph if desired.
7. end while
8. Undo any alterations made in Steps 1 and 6.
```

This is the most general method available, since any global method and any local

method can be used in this algorithm and can both be iterative methods. It is the "state of the art" partitioning algorithm [12, 63, 29], commonly used with SPECT along with graph coarsening as the global partitioning algorithm and KL as the local cleanup algorithm. Karypis and Kumar [40, 39] investigated and implemented greedy initial partitioning schemes which they state work better than SPECT.

1.5 A New Model for the Data Distribution Problem

Although mathematically pleasing, the number of edges cut does not provide even a qualitatively correct relative measure of the costs of a data distribution, as will be shown in Chapter 4. The AET (approximate execution time) metric presented below is a proposed replacement for the edges cut metric as the measurement of a partitioning's quality for the data distribution problem.

The AET metric accounts for:

1. The parallel machine MA executing the linear system solver using a simple model of the parallel processing environment described in Chapter 2.

2. The linear solver algorithm or *solver* and its data executed on the parallel computer. Representation of the solver should minimally include the amount and

range of data values, kernel operations of the solver and the execution time of those kernels. Most current work in solver estimation targets solvers with limited applicability, such as direct methods [27] or simple iterative methods with block diagonal preconditioners [22]. Unfortunately, in practice, those solver algorithms either require more memory than is currently available, or simply fail to converge for realistic 3D problems. Some popular solver algorithms, whose performance is assumed to be dominated by kernel execution times, are described in Chapter 3.

3. The distribution of data, particularly A, among processors of the computer; that is, how the data is partitioned and mapped.

The estimates of these component parts are combined using a simulation routine described in Chapter 4 to determine a TPI (time per iteration) estimate for the solver.

However, if the AET metric is to be of practical use, it should take significantly less time to calculate than the actual execution time. The ultimate cost function for any program is the execution time of that program on the user's input and hardware. However, running a program using various arrangements of the data on a parallel computer to find the best partitioning can be more expensive than running it once with a bad data arrangement.

It is proposed that this AET heuristic replace the edges cut metric calculation in a variation of the KL (Kernighan-Lin) local graph partitioning algorithm. KL is a natural choice as it only considers pairwise interchanges of nodes based on comparison the absolute values of the a cost function. The faster F-M variant of KL uses the fact that the edges cut function returns integer values to optimize the algorithm, and does not seem to be easily modifiable to handle the floating point AET values. AET does not seem readily applicable to any global partitioning scheme. Other options would include replacing the edges cut or other metric with the AET calculation in a SA algorithm, but since SA does not have a good cost vs. performance ratio, it is not a likely candidate.

The main modifications to the KL algorithm would entail first estimating each processor's exeuction time. The maximum and minimum AET values would be calculated and, if the relative difference between the AET values were small enough or if too many data interchanges had been made, the algorithm would terminate. Then, block rows would be moved from the slowest processor to the fastest processor by updating the block row assignment vector with the interchanged block rows (more than one should be moved at a time for efficiency) and the matrix would be rearranged using the new assignment. This solution likely will be slow, especially if the block rows are moved one at a time. An obvious optimization to the proposed solution is to move multiple block rows at a time, either a fixed number at each iteration or some

proportion of the matrix size. The need to map partition sets to processors would be removed using the AET metric, since the AET metric accounts for the parallel machine running the solver which will use the data distribution.

Clearly, this solution is not a new data distribution *algorithm*, but instead a new *model* of the data distribution problem. This solution's applicability is tailored to this problem, but could be applied to any problem whose performance is dominated by a small number of known functions by merely changing the kernel cost functions. Instead of being a new algorithm, the proposed solution is a generalization from the generic edges cut cost function to one using a general cost function, and the proposed use of KL could be replaced by any graph partitioning algorithm that can handle a non-monotonic, real valued cost function.

1.6 Thesis Organization

Chapter 2 describes the hardware and software parallel environments used in the thesis and presents a parallel program modeling strategy based on modeling kernels. Chapter 3 outlines the HMPE solver used for thesis results. Chapter 4 gives computational results of the AET metric calculation as part of HMPE, which indicates the AET metric is the best currently available predictor. Finally, Chapter 5 summarizes the thesis and suggests future directions for this work.

2

Parallel Computational Environments

This chapter provides a listing of the computer architecture classes and the basic hardware specifications for the parallel computers used for experimentation and results. The Message Passing Interface (MPI) Version 1.0 standard for parallel communication and synchronization is summarized. Vendor implementations of MPI are the parallel communications libraries used in the code. Finally, a performance model for a parallel program is defined.

2.1 Parallel Computer Architectures

Two parallel architecture types – **shared memory** and **distributed memory** – are used for results presented in the disseration. They are the dominant architecture types today, but new hybrid architecture classes are becoming commercially available.

2.1.1 Shared Memory Computers

A shared memory computer has multiple CPUs tied together with some interconnection network, typically a bus, and a large common global memory. Interprocessor communication is performed by global memory reads and writes. For shared memory computers, memory performance directly affects communications performance. Generally, shared memory computers have a relatively small number of CPUs to minimize shared memory contention. They have better communications performance for those CPUs and are generally easier systems for code development than distributed memory computers.

An example of this architecture type is the SGI Power Challenge [54], which is equipped with either of the MIPS R8000 or MIPS R10000 CPU chips and up to 16 GB of global memory, connected by a 1.2 GB/second common bus that uses piggy-back reads. The Power Challenge used for the disseration results has 10 75 MHz R8000 chips and a three tier RAM hierarchy: a 16 KB cache on the integer unit, which is

unused during floating point loads and stores [54, Ch. 3, p. 47], a 4 MB second-level off-chip "data streaming" cache, and a 2 GB global shared memory.

For larger problems, **shared memory array** computers can be used. A shared memory array is several shared memory computers by a relatively fast inter-processor network. An example is the SGI Power Challenge Array, which is a collection of two to eight SGI Power Challenges (each of which can have up to 36 R10000 CPUs) tied together by a high-performance network, based on either HiPPi, ATM, FDDI, or Ethernet.[1]

2.1.2 Distributed Memory Computers

Distributed memory computers have two or more computers or **nodes**, each with their own local memory, tied together by a fast, private interconnection network. Interprocessor communication is performed by sending messages on the interconnection network between nodes. Distributed memory computers typically have more processors than shared memory computers, but each node is generally less powerful than the average processor of a shared memory computer. Code development is generally more difficult for a distributed memory computer, since inter-processor communication must be done using message passing routine calls, instead of a simple

[1]This information is from the Power Challenge Array WWW Page `http://www.sgi.com/Products/hardware/Power/pc_array/pc_array.html`.

memory read/write as is possible on shared memory computers.

The distributed computers used for testing the dissertation code are the IBM SP2 and the Intel Paragon. The SP2 can be configured to use between 2 and 512 CPUs. The SP-2 used [46] for the results presented below has 24 RS/6000 SP Thin-2 Node processors. Each processor has a 67MHz clock with a two- or three-tier RAM hierarchy, depending on configuration. The first level is a 128 KB data cache (used for both integer and floating point data, unlike the Power Challenge). 12 of the 24 SP2 nodes have a second level 2 MB data cache. All 24 nodes have at least a 128 MB main memory – 4 nodes have 512 MB main memories, 8 have 256 MB main memories, and the remainder have 128 MB main memories. They are interconnected using a version SP High Performance Switch, a multi-stage packet switched Omega switch providing about 80 MB/second bandwidth with a minimum of 4 paths between nodes.

The Intel Paragon used [14] is a model XP/S-A7 with 92 general purpose and 4 I/O and service nodes connected in a 12×8 mesh. Each node has an Intel 50 MHz i860 XP microprocessor, a 16 MB local memory with roughly 6 MB occupied by the OSF/1 Version 1.0.4, R1_3 operating system, and two (data and instruction) 16 KB caches. The peak memory-to-memory transfer rate is 400 MB/s and the peak transfer rate between the data cache and the FPU is 1.2 GB/s. For communication, each node has a separate Mesh Router Chip dedicated to message traffic along with a Network Interface Controller chip to connect the Mesh Router Chip with the XP's

memory. The peak transfer rate both between two nodes and between the network and the XP's memory is 200 MB/s full duplex.

A **distributed shared memory** processor replaces each node of a distributed memory computer with a shared memory multiprocessor. Both of these types of hybrid shared/distributed memory computers are expected to be more widespread in the future.

2.2 The MPI Standard

MPI is a communication and synchronization interface standard [56, 18] for parallel computers based on message passing used in the HMPE solver. It is the dominant parallel computing interface standard, replacing Parallel Virtual Machine (PVM) [1].

2.2.1 What is MPI?

MPI is a standard defining a parallel communications paradigm – it is not a implementation. Standardization allows for portable code between different parallel hardware platforms and between different MPI implementations. The wide acceptance of the standard is due in part to its readability [56]. Free implementations

[59, 45] are available that run on most parallel platforms along with native implementations for Cray, IBM, and SGI computers,[2] to name a few. The MPI standards process is working on a second version (MPI-2) of the standard, which should be available in 1997 [57].

2.2.2 Features of the MPI Version 1.0 Standard

The current version of the standard concentrates on providing a variety of message passing models for point-to-point communications. The standard defines simple models of process creation, destruction, synchronization and collective communication. MPI allows the use of both pre-defined and user-defined data types and various communications topologies. Both FORTRAN 77 and C function bindings with language independent semantics are provided. This requires the specification of about 125 different functions.

Though the standard is complex, the HMPE solver uses only eleven MPI functions listed in Table 2.1, and five of those functions are only used in MPI initialization and finalization. The MPI standard is both large and small, in that it allows for a wide variety of communications possibilities, but does not require a large number of functions to get a large parallel program to run.

[2]This source of this information is the Argonne National Laboratories MPI WWW Page `http://www.mcs.anl.gov/mpi/index.html` and IBM SP-2 on-line documentation.

Table 2.1: MPI Functions Used in the HMPE Solver

MPI_Barrier	MPI_Isend	MPI_Irecv	MPI_Test
MPI_Allreduce	MPI_Bcast	MPI_Init	MPI_Comm_dup
MPI_Comm_size	MPI_Comm_rank	MPI_Finalize	

MPI implements a simple process model. Version 1.0 does not allow for process creation or destruction after MPI initialization. It does not describe standards for shared memory or I/O operations, debugging tools, or even a uniform command line interface. Many of these problems and oversights in Version 1.0 are addressed in the draft of the MPI-2 standard [57]. The model described below assumes the use of MPI Version 1.0.

The use of a message passing paradigm indicates the standard was written with distributed memory machines in mind. Even on shared memory machines, the advantage of being able to code to a well defined and widely accepted standard generally outweighs any performance disadvantages. This advantage is increased if a shared memory code is to be ported and run on a shared memory array.

2.3 A Model for Parallel Computer Performance

A model of parallel computer performance is described that accounts for the greater part of performance variations due to changes in the parallel environment for

a specific type of parallel programs. The model does not require detailed hardware specifications or assembly-level run-time traces. It is fundamentally influenced by the ideas given in [36, pp. 30-33] to give a model that combines analysis, simulation, and direct measurements.

2.3.1 Machine Definition

A **machine** $\mathcal{M}$ running a parallel program Δ is defined to be the total parallel execution environment used to execute Δ. $\mathcal{M}$ is assumed to be running $\mathcal{P}$ **processes** $p_1, p_2, \ldots p_{\mathcal{P}}$ in parallel using $\mathcal{N}$ distinct hardware **processor types** $H_1, H_2, \ldots, H_{\mathcal{N}}$. Note that a machine consists of one or more (parallel) computers. Let $NCPUS(H_i)$ be the number of CPUs (central processing units) utilizing processor type $H_i, 1 \leq i \leq \mathcal{N}$ and let $N = \Sigma_{i=1}^{\mathcal{N}} NCPUS(H_i)$. Under the assumption that each CPU runs at most one process, $\mathcal{P} \leq N$. Figure 2.1 shows a simple example of these relationships.

2.3.2 Description of a Process

A working definition of a process is "code in motion": that is, a **static** set of instructions operating on some **data** executing in a **dynamic** environment. The process description uses a **kernel-based** model of the static components of a process

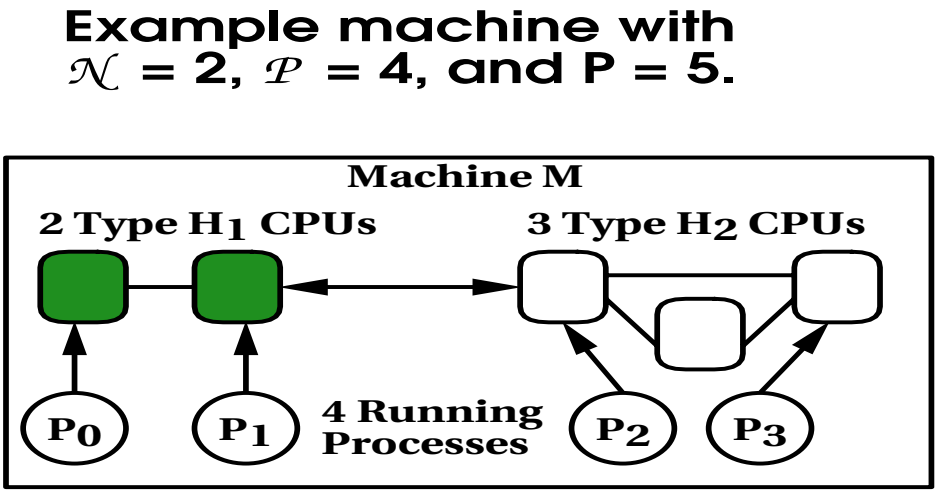

Figure 2.1: Parallel Machine Model Example. $\mathcal{N} = 2$, $N = 5$ with $NCPUS(H_1) = 2$ and $NCPUS(H_2) = 3$, and $\mathcal{P} = 4$.

operating in a fixed environment over some data and the **process state** in a dynamic computational environment.

Δ is viewed as a combination of a fixed number K_{hi} of **high-level kernels** operating on some input data. Each process $p_l, 1 \leq l \leq \mathcal{P}$, running Δ is assumed to be coded as a list high-level kernel calls. A high-level kernel is assumed to consist of calls to one or more of K **low-level kernels**, each running on some input. This breakdown of high and low level kernels is based on a coding strategy described in [47] of parallel high-level kernels comprised of calls to (uniprocessor) low-level kernels. It is also used in the the HMPE solver described in Chapter 3. p_l's execution trace can be stated as the series of low-level kernel calls that make up the list of Δ's execution of high-level kernel calls on some input.

Each of the p_l's process state includes the basic information needed to model a

uniprocessor: cache contents and size, processor clock, and a program counter. It is assumed that all data is resident in either cache or in main memory – the process state in this model does not include I/O. It also includes the communications/synchronization status of a particular CPU within the parallel computer network.

2.3.3 Process Performance Modeling

Δ's performance model assumes each of the K low-level kernels has a performance model for each processor type $H_i; 1 \leq i \leq \mathcal{N}$. In this dissertation, the low-level kernel performance model or **timing model** $TM^j_{H_i}$ for kernel $K_j, 1 \leq j \leq K$ running on a processor of type H_i is based on analysis of repeated timings of K_j on H_i. Chapter 4 further describes timing models. The **architecture** $AR_i, 1 \leq i \leq \mathcal{N}$, is the collection of all K_j timing models for processor type H_i, since from a performance point of view, the timing models for a processor type or architecture *are* the architecture.

A timing model for kernel K_j takes the **sizes** of each of K_j's inputs as its input and returns an **initial estimate** $\mathcal{E}_{init}$ of K_j's performance on H_i; where the size is the amount of memory required to store each input. For example, if K_j is called with two parameters x and y, with sizes $|x|$ and $|y|$ respectively, $\mathcal{E}_{init}$ for the kernel call $K_j(x, y)$ is $\mathcal{E}_{init} = TM^j_{H_i}(|x|, |y|)$.

To implement this modeling strategy, a **simulator** is used to maintain the flow of p_l's execution, determine K_j's input sizes for each low level kernel K_j, and maintain

p_l's process state. Simulation is required to determine both the K_j's input sizes and p_l's process state as neither can be known *a priori* given general inputs or different computational environments. The simulator combines $\mathcal{E}_{init}$ with the cache hit ratio $\mathcal{H}$ and sychronization state maintained in the process state to get a **final estimate** $\mathcal{E}_{final}$ for k. The process state's clock value $\mathcal{E}$ is updated by adding $\mathcal{E}_{final}$ to the previous clock value, so $\mathcal{E} = \Sigma\mathcal{E}_{final}$. Since Δ is assumed to be made up of a list of high-level kernels, which are in turn made up of low-level kernels, the estimate of the time p_l's spends running Δ is the sum of the low-level kernel estimates $\mathcal{E}$.

2.3.4 Related Work

Blau's [6] 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 [2] and Xu, Zhang, and Sun [62] 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. Both model fairly simple programs whose task graphs are known in advance.

The static estimation portions of this strategy (i.e. the timing models) can be viewed as a case of *abstract interpretation* as applied to resource estimation for parallel programs. Abstract interpretation [15] is a general term for a compile-time analysis of a function or program used for estimating its properties. The term originated with researchers in functional and logic programming interested in type checking and semantics. Other program resources can be determined using abstract interpretation techniques, such as the memory utilization of the matrix-matrix multiplication operation[42].

2.3.5 Critique of this Performance Modeling Strategy

The performance modeling strategy described above is applicable to systems with a moderate number of pre-defined kernel operations. It is most applicable for systems that are comprised of small numbers of widely known and used kernels, allowing for independent verification of estimation results. While it is not as general as Adve's or Xu, Zhang, and Sun's system, specialization allows this modeling strategy to provide

accurate results for a program which need not be completely specified ahead of time.

Clearly not all programs are easily broken down into a series of kernels. The timing data required for the kernel estimation functions can be difficult or impossible to generate, especially for new systems or systems with kernels that exhibit random performance characteristics. There is nothing in this modeling strategy that should restrict its use on uniprocessor codes that can be written as a series of (high and) low-level kernel calls. However, given a program coded in this fashion, use of this modeling strategy should provide for reasonably accurate results at a reasonable cost.

2.4 Summary

A brief overview of the parallel hardware and software environments used in the remainder of the dissertation was presented. A method for modeling parallel environments was described based on a combination of static and dynamic models, which will be extended in later chapters. The next chapter describes the HMPE linear system solver, which is used as a test case for this modeling strategy.

3

The HMPE Linear System Solver

3.1 Introduction

A **linear system solver** or **solver** for short is a program that solves the set of linear equations $Ax = b$, where A is an $n \times n$ (or **order** n) structurally symmetric matrix and x and b are $n \times 1$ vectors. A contribution of this dissertation is the HMPE parallel preconditioned iterative solver package. HMPE implements the CGSTAB [58] and GMRES [51] solver algorithms, the block Jacobi [5] and block SSOR [61] preconditioning algorithms, and five factorization schemes for the diagonal (intraprocessor) blocks of A. HMPE is implemented in C++ using the MPI [56] communication library and has been tested on the SGI Power Challenge [54] shared memory and the IBM SP-2 [35] and Intel Paragon [14] distributed memory computers.

The solvers in HMPE are coded in terms of a small number of kernels listed in

Appendix A. In HMPE, kernels are further broken down into **high-level kernels** and **low-level kernels**, which allows HMPE to use the modeling strategy described in Section 2.3. High-level kernels provide the interface between the mathematical solver specification and details of the computational environment and data structures, which are handled by low-level kernels. See Appendix B for the implementation of HMPE's solver algorithms in terms of the high-level kernels listed in Appendix A. HMPE does not implement any new preconditioners or solver algorithms; it was designed to provide an example system for modeling based on the best currently available iterative solvers. HMPE performance results are given in Chapter 4.

The rest of this chapter is organized as follows: Section 3.2 describes some preconditioned linear system solution methods, primarily taken from [5, 50]. Section 3.3 describes some of the matrix data structures commonly used, Section 3.4 defines and describes the use of computational kernels in HMPE. Section 3.5 concludes and summarizes the chapter.

3.2 Linear System Solution Methods

There are two classes of linear systems solution methods: **direct** and **iterative** methods [20, 50]. Direct methods **factor** A into $A = LU$, where L is lower triangular, and U is upper triangular and then solve $Ax = (LU)x = b$. During factorization, the

entries of L and U overwrite the entries of A. The factorization process can create non-zero entries for L and U that were not in A, so A could require in the worst case a **dense** data structure. Dense data structures allocate memory for each potential entry of A (i.e. $(n \times n)$ memory locations). This $\mathcal{O}(n^2)$ storage requirement is clearly becomes impractical as problem sizes increase.

Most matrices from PDE discretizations contain mostly zero entries. **Sparse matrix** storage and numerical techniques are designed to store and operate only on the non-zero entries of A. Sparse matrix techniques are usually associated with **iterative solvers**. An iterative solver is one that makes successive approximations to a solution; that is, it iterates toward convergence between the solution guess and the real solution. Iterative methods do not update the entries of A, but only require the result w of $Ad = w$, and for some solvers, $A^T d = w$, where A^T is the transpose of A. However, no single iterative solver guarantees convergence for a general A. Direct methods that operate on sparse matrices have been developed [20], but this dissertation will concentrate on their use as preconditioning methods for iterative solvers.

Preconditioning is used to increase the probability an iterative solver will converge by transforming A to an easier system to solve. Preconditioning is commonly performed by using a second matrix, M, known as the **preconditioning matrix**. HMPE supports two block preconditioning algorithms. The implementation assumes

each diagonal block is factored using **incomplete** LU factorization methods first described in [61].

3.2.1 Iterative Solvers

Three solver algorithms are described below: the Conjugate Gradient (CG) [32], Generalized Minimum Residual (GMRES) [51], and Conjugate Gradient Stabilized (CGSTAB) [58]. Iterative solvers generate **residual sequences** that hopefully converge toward the true solution x. These sequences are written as $\{r_i\}, i \geq 0$. Given a matrix A and some initial guess x_0 to the solution of $Ax = b$, the initial error or **residual vector** r_0 is $r_0 = b - Ax_0$, and each succeeding iteration i generates a new r_i. **Convergence** is reached when $\|r_i\| < \epsilon\|r_0\|$; typical choices for ϵ range from 10^{-6} to 10^{-10}.

All three algorithms are based on projecting sequences of vectors onto Krylov subspaces. The first two methods are based on the Arnoldi orthogonalization technique, and CGSTAB is based on a transpose-free variant of Lanczos bi-orthogonalization. The Arnoldi technique applies the modified Gram-Schmidt orthogonalization method to the Krylov sequence $\{A^k r_0\}, k = 0, 1, \ldots$. The transpose-free variation of Lanczos bi-orthogonalization used in CGSTAB generates residual vectors of the form $r_k = \psi_k(A)\phi_k(A)r_0$, where ψ_k and ϕ_k are polynomials in A. See [50] for a good reference on iterative solvers.

3.2.1.1 The Conjugate Gradient (CG) Algorithm . The CG algorithm [32] assumes the input matrix A and the preconditioning matrix M are both symmetric positive definite (SPD). The method generates approximations to the solution x_i, corresponding residual vectors r_i , and **search vectors** d_i. Many approximations may be needed, but the number of vectors that need to be stored is small.

Each CG iteration updates the solution vector x_i by a multiple α_i of the search vector d_i: $x_i = x_{i-1} + \alpha_i d_i$. The residual r_i is updated in the Arnoldi fashion by subtracting $\alpha_i(Ad_i)$ or $r_{i+1} = r_i - \alpha_i(Ad_i)$, where $\alpha_i = \|r_i\|^2/(d_i, Ad_i)$ minimizes ${r_i}^T A^{-1} r_i$. Search directions are updated using $d_i = r_i + \beta d_{i-1}$, where $\beta = (\|r_i\|_2)^2/(\|r_{i-1}\|_2)^2$. It can be shown that this choice of β makes d_i orthogonal to $Ad_j, j = 0, 1, \ldots, i-1$ and r_i orthogonal to $r_j, j = 0, 1, \ldots, i-1$. See [52] for a lucid description of the CG algorithm, its underlying principles, and proofs of these results for the unpreconditioned algorithm.

The complete unpreconditioned CG algorithm is stated below:

Initialize:

$x_0 \;=\; given$

$r_0 \;=\; b \;-\; A\, x_0$

$d_1 \;=\; r_0$

$i \;=\; 1$

$\gamma_0 \;=\; r_0^T\; r_0$

End Initialization

while *($\gamma_{i-1} > tol$)* **and** *($i \leq maxits$))*

$w = A\ d_i$

$\tau = w^T\ d_i$

$\alpha_i = \gamma_{i-1}\ /\ \tau$

$x_i = x_{i-1} + \alpha_i\ d_i$

$r_i = r_{i-1} - \alpha_i\ w$

$\tau = r^T\ r$

$\beta_i = \tau\ /\ \gamma_{i-1}$

$\gamma_i = \tau$

$d_i = r_i + \beta\ d_i$

$i = i + 1$

endwhile

end

Because A is assumed to be SPD, CG can maintain three properties throughout its run: construction of an orthogonal basis for the Krylov subspace span $[r_0, Ar_0, \ldots, A^{i-1}r_0]$, minimization of the error terms, and short recurrences, which leads to storing a small number of vectors. The implementation of the unpreconditioned CG algorithm above uses only three additional vectors: d, r, and w. One of the three properties of orthogonality, minimization, and short recurrences must be

given up for all but trivial non-SPD matrices [24]. Both the GMRES and CGSTAB algorithms described below are applicable for general, nonsymmetric, non positive definite matrices; GMRES gives up short recurrences, and CGSTAB gives up the minimization property.

3.2.1.2 The GMRES Algorithm. In CG, the residuals form an orthogonal basis for the Krylov subspace span $[r_0, Ar_0, \ldots, A^S r_0]$. GMRES forms this basis explicitly using a modified Gram-Schmidt orthogonalization procedure:

$w_i \ = \ A \ y_i$

for $k = 1$ **to** S

$\quad w_i \ = \ w_i \ - \ (w_i, \ y_k) \ y_k$

endfor

$y_{i+1} \ = \ w_i \ / \ \| w_i \|$

After S steps, this creates a $S+1 \times S$ upper Hessenberg matrix

$$\bar{H}_S = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \dots & & & h_{1K} \\ h_{21} & h_{22} & h_{23} & \dots & & & h_{2K} \\ & h_{32} & h_{33} & h_{34} & \dots & & h_{3K} \\ & & \ddots & \ddots & \ddots & & \\ & & & \ddots & \ddots & \ddots & \\ & & & & & h_{K,K-1} & h_{KK} \\ & & & & & & h_{K+1,K} \end{bmatrix} \tag{3.1}$$

where $h_{ii} = \|w_i\|$ and $h_{ik} = (w_i, y_k)$. Let $Y_S = [y_1, y_2, \dots y_S]$. The next iterate is given by $x_i = x_0 + Y_S y$, where y minimizes the residual norm $\|r_i\| = \|b - Ax_i\|$. By explicitly maintaining S vectors, GMRES gives up short recurrences while keeping the orthogonality and minimization properties of CG. To keep storage requirements reasonable, GMRES is typically restarted after m steps, giving the GMRES(m) algorithm. Appendix B.3 has the preconditioned algorithm statement of GMRES(S).

3.2.1.3 Conjugate Gradient Stabilized (CGSTAB) . CGSTAB uses short recurrences and maintains a form of the orthogonality constraints called bi-orthogonality, but it does not keep the residual minimization property. The fast and smoothly converging variant of the algorithm [58] is listed below.

Initialize:

$r = b - A\ x$

$p = r$

$v = q = 0$

$\omega_h = \beta = \alpha = 1$

$k = 1$

End Initialization

while *((*$\| r \| > tol$*)* **and** *(*$k < maxits$*))*

$\beta_h = p^T r;\ \omega = (\beta_h\ \omega_h)/(\beta\ \alpha)$

$\beta = \beta_h$

$q = r + \omega(q - \alpha\ v)$

$v = A\ q$

$\tau = p^T v$

$\omega_h = \beta_h\ /\ \tau$

$s = r - \omega_h\ v$

$t = A\ s$

$\sigma = t^T s;\ \tau = t^T t$

$\alpha = \sigma /\ \tau$

$x = x + \omega_h q + \alpha\ s$

$r = s - \alpha\ t$

$k = k + 1$

endwhile

The algorithm computes the sequence of residual vectors $r_i = \psi_i(A)\phi_i(A)r_0$, where ψ_i and ϕ_i are i^{th} degree polynomials in A. The ϕ_i polynomial is constructed to maintain orthogonality between the residual iterates and the span of the Krylov subspace; ω_h is the most recently calculated coefficient of ϕ_i. The ψ_i polynomial is used to stabilize (smooth) convergence; α is the most recently calculated coefficient of ψ_i. and is represented by the α variable. Other polynomials of the form of ψ_i can also be used, and several variants are evolving that attempt to prevent breakdowns, minimize other error norms, etc. See Appendix B.2 for the statement of the algorithm listed above in terms of high-level kernels.

3.2.2 Preconditioning

The goal of preconditioning is transform the system $Ax = b$ to an system that takes less time to solve. The general preconditioned system is $M_L^{-1}AM_R^{-1}x = M_L^{-1}b$, where M_L^{-1}, the **left preconditioner**, and M_R^{-1}, the **right preconditioner**, are both nonsingular. We will concentrate on left preconditioning with $M = M_L$. This reduces the general form of the preconditioned system to be solved to $M^{-1}Ax = M^{-1}b$. If $M \approx A$, then $M^{-1}b = M^{-1}Ax \approx A^{-1}Ax = x$, then $M^{-1}b = x$, so the closer M approximates A, the better M acts as a preconditioner.

As implemented, preconditioning is a two phase process. Preconditioning **setup**

involves the generation of M from A. M is assumed to be generated using an **incomplete LU factorization** of A; that is, $M = LU$ where L is a unit lower triangular matrix and U is an upper triangular matrix. There are a variety of methods to incompletely factor a matrix [50, pp. 265-298], but only the **ILU(s)** factorization method was used as described below. Setup costs are not included in the timing results.

Preconditioner **application** is the calculation of $z = M^{-1}y$ by solving the system $Mz = y$ for z. The preconditioner is applied to the initial residual r_0, since the initial error $r_0 = b - Ax_0$ has been transformed to $M^{-1}r_0 = M^{-1}(b - Ax_0)$. It is also applied each time a matrix-vector multiplication is required by the unpreconditioned solver. That is, if the unpreconditioned solver requires $z = Av$, then the corresponding preconditioned solver requires $z = M^{-1}(Av)$. A solver code first calculates $d = Av$ and then the preconditioner $z = M^{-1}d$ is applied. Using the fact $z = M^{-1}d = (U^{-1}L^{-1})d = U^{-1}(L^{-1}d)$, the preconditioner application $z = M^{-1}d$ is performed by first solving $Ly = d$ for y and then solving $Uz = y$ for z.

The preconditioned CG algorithm is presented as an example. The version below applies the preconditioner to the residual vector r on each iteration instead of directly following the matrix-vector multiplication in the solver loop.

Initialize:

$x_0 \;=\; given$

$r_0 \;=\; b \;-\; A\,x_0$

$solve\ M\ z_0\ =\ r_0\ for\ z_0$

$d_1\ =\ r_0$

$i\ =\ 1$

$\gamma_0\ =\ r_0^T\ r_0$

End Initialization

while $((\gamma_{i-1}\ >\ tol)$ **and** $(i\ \leq\ maxits))$

$w\ =\ A\ d_i$

$\tau\ =\ w^T\ d_i$

$\alpha_i\ =\ \gamma_{i-1}\ /\ \tau$

$x_i\ =\ x_{i-1}\ +\ \alpha_i\ d_i$

$r_i\ =\ r_{i-1}\ -\ \alpha_i\ w$

$solve\ M\ z_i\ =\ r_i\ for\ z_i$

$\tau\ =\ r^T\ r$

$\beta_i\ =\ \tau\ /\ \gamma_{i-1}$

$\gamma_i\ =\ \tau$

$d_i\ =\ r_i\ +\ \beta\ d_i$

$i\ =\ i\ +\ 1$

endwhile

Parallel solvers partition A and M into blocks; in HMPE, each process owns one or

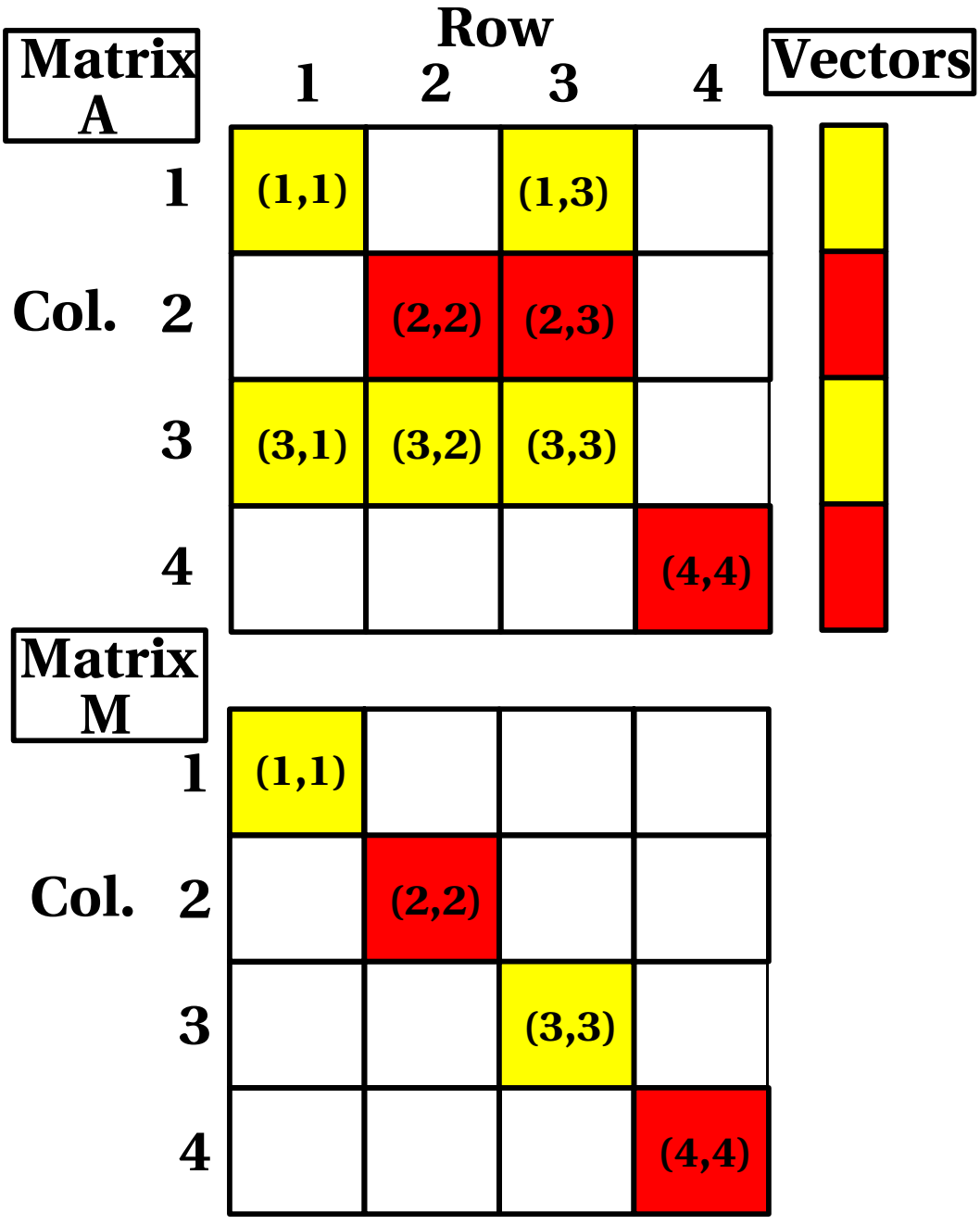

Figure 3.1: This example of HMPE block ownership represents a 4×4 block matrix A and a 4×1 vector, both of whose blocks are cyclically distributed among two processes and the corresponding preconditioning matrix M. The differently colored blocks are owned by different processes.

more block rows of both matrices and the corresponding vector blocks. Each diagonal block of A is incompletely factored into its L and U component triangular matrices during setup and the factored diagonal blocks are stored in M. See Figure 3.1 for a simple example of process block ownership for A, M, and vectors. For preconditioner application, HMPE allows the use of either the BSSOR (Block Symmetric Successive Over-Relaxation) and BDIAG (Block DIAGonal or block Jacobi) preconditioners.

3.2.2.1 ILU(s) Partial Factorization. An LU factorization is performed by factoring A into L and U, where L is unit lower triangular and U upper triangular. LU factorization is based on Gaussian elimination, while retaining the multipliers in the L matrix and the pivots are scaled and stored along the diagonal of the factored matrix [44]. To outline the LU factorization process, let $A = L' + D + U'$, with L' the strictly lower triangular part of A, U' the strictly upper triangular part of A, and D the diagonal of A. Let $a_{ij}, 1 \leq i, j \leq n$ be an entry of A and assume $a_{ii} \neq 0, i = 1, 2, \ldots n$. One version of Gaussian elimination for a dense A without pivoting progresses along each row $i = 1, 2, 3 \ldots n$ of A. For each (non-zero) entry in row i of L', $a_{ik}, k = 1, 2, \ldots n-1$, scale and update the entry $a_{ik} = a_{ik}/a_{kk}$. Then, for each (non-zero) entry in row i of U', $a_{ij}, j = i+1, i+2, \ldots n$, update a_{ij} with

$$a_{ij} = a_{ij} - a_{ik}\ a_{kj} \tag{3.2}$$

If $a_{ij} = 0$ and $a_{kj} \neq 0$ before a_{ij}'s update, a_{ij} will become a **fill** or new non-zero entry. At the end of the process, let $L = L' + I$ and let $U = D + U'$, and $A = LU$.

If some of the fill entries are discarded as part of the LU factorization process, an **incomplete LU (ILU) factorization** of A would be generated. For the ILU(s) factorization process, the decision to keep a fill entry is determined by an entry's **fill level** lev_{ij}. The definition [50, p.280] of the initial fill level lev_{ij} of an element a_{ij} of a sparse matrix A is

$$lev_{ij} = \begin{cases} 0 & \text{if } a_{ij} \neq 0 \text{ or } i = j \\ \infty & \text{otherwise} \end{cases}$$

Every time a_{ij} is modified by Equation 3.2 during the LU factorization process, lev_{ij} is updated using $lev_{ij} = \min(lev_{ij}, lev_{ik} + lev_{kj} + 1)$. Only if $lev_{ij} \leq s$ is the entry kept. For the special case $s = 0$, the factored matrix will have the same sparsity pattern as the unfactored matrix.

3.2.2.2 The Block Jacobi/Diagonal (BDIAG) Preconditioner . Since a preconditioner M only approximates A, one simple approximation is to let $M = D$. Applying this idea to a block matrix leads to the BDIAG preconditioner. To generate M, the diagonal blocks of A are factored using ILU(s) and stored in M. BDIAG preconditioner application requires one lower and one upper triangular solve for each diagonal block of A. This corresponds to a simple loop over the diagonal blocks of A; see Appendix B for the preconditioner's algorithm statement. This preconditioner

is perfectly block parallelizable since all blocks of M are independent. Each solver iteration with BDIAG is relatively fast, but BDIAG often takes a large number of iterations or fails to converge.

3.2.2.3 The Block SSOR (BSSOR) Preconditioner. The BSSOR preconditioner assumes $M = L'_{block} + D_{block} + U'_{block}$, where D_{block} consists of the factored diagonal blocks of A, L'_{block} is the block lower triangular portion of A, and U'_{block} is the block upper triangular portion of A. A is assumed a square NBR by NBR block matrix, where NBR is the number of block rows of A.

The BSSOR algorithm is a lower block triangular solve followed by an upper triangular solve. Since M need not store the off-diagonal blocks of A, both A and M are parameters for the BSSOR preconditioner algorithm performing $y = M^{-1}z$:

$BSSOR(A,\ M,\ z,\ y)$

begin

Comment: *Lower triangular solve*

$y\ =\ z$

$d\ =\ 0$

for $k\ =\ 1$ **to** NBR

for *each block* $A_{row,col}$ *in* A

if *((col* $=\ (k-1)$*)* **and** *(row* $\geq\ k$*))* **then**

$d_{row}\ =\ d_{row}\ +\ A_{row,col}\ y_{col}$

endif

endfor

$y_k \;=\; d_k \;-\; y_k$

$y_k \;=\; M_k^{-1}\; y_k$

`BROADCAST(`y_k`)`

endfor

Comment: `Upper triangular solve`

$d \;=\; 0$

$d_{NBR} \;=\; y_{NBR}$

for $k \;=\; NBR$ **to** `1` step -1

for `each block` $A_{row,col}$ `in` A

if `((`$col \;==\; (k+1)$`)` **and** `(`$row \;<=\; k$`))` **then**

$d_{row} \;=\; d_{row} \;+\; A_{row,col}\; d_{col}$

endif

endfor

$d_k \;=\; M_k^{-1}\; d_k$

$d_k \;=\; y_k \;-\; d_k$

`BROADCAST(`d_k`)`

endfor

$y \;=\; t$

end

This algorithm is also listed in Appendix B in terms of high-level kernels. If BSSOR is run on in a multiprocessor environment, the vector block most recently calculated must be sent to all processors requiring it at the points marked `BROADCAST` in the algorithm. This requirement forces synchronization between the processes, which usually greatly increases BSSOR's parallel run-time overhead. Since BSSOR is a fairly robust preconditioner, it may be required to solve difficult systems regardless of cost. Parallelizable versions of BSSOR based on using low rank approximations of off-diagonal blocks combine the robustness of BSSOR with the parallelism of BDIAG [8].

3.3 Data Structures

As stated in Section 3.1, there are two usual ways to store a matrix: dense and sparse. Dense structures are easier to use from a programmer's point of view, but use more memory and CPU cycles in computing the result zero elements. For parallel computing, there is another consideration: how to store the matrix on multiple processors. The usual solution is to divide the matrix into **matrix blocks** and have each processor store either **block rows** or **block columns** of the matrix.

There are a wide variety of sparse data structures, of which three of the most

common – coordinate-wise (COO), compressed sparse row (CSR), and modified sparse row (MSR) – will be described below. In HMPE, the matrix data structure is stored in two levels: a global level and a block level. The global level is implemented using a HyperMatrix data structure described below. Each block of A is stored in CSR format, and the factored blocks of M are stored in MSR format.

3.3.1 Local Data Structures

We define **local** data structures and operations as those which pertain to *individual* matrix and vector blocks. **Global** data structures and operations as those which pertain to the *whole* distributed matrix and vectors. For the next three sections, the matrix A_{ex} is used as an example:

$$A_{ex} = \begin{bmatrix} 11 & 12 & 13 & 14 & & \\ 21 & 22 & 23 & & 25 & \\ & 32 & 33 & & & \\ & & & 44 & & 46 \\ 51 & & & 54 & & \\ & & 63 & & 65 & 66 \end{bmatrix}, \tag{3.3}$$

where the missing entries to be zeros. A_{ex} is of order $n = 6$ rows and columns or is of **order** $n = 6$ and the number of non-zero entries NNZ for $A_{ex} = 17$.

Table 3.1: COO Storage Layout for A_{ex}

Array Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`row_inds`	1	1	1	1	2	2	2	2	3	3	4	4	5	5
`col_inds`	1	2	3	4	1	2	3	5	2	3	4	6	1	4
`values`	11	12	13	14	21	22	23	25	32	33	44	46	51	54

Array Index	15	16	17
`row_inds`	6	6	6
`col_inds`	3	5	6
`values`	63	65	66

3.3.1.1 Coordinate-Wise (COO) Data Structure. A simple sparse matrix data structure is the coordinate-wise (COO) data structure. The matrix is stored as three coordinated arrays: `row_inds`, `col_inds`, and `values`. The i^{th} entry of each array holds the i^{th} row index, column index, and data value, respectively, for each non-zero entry of the matrix; the entries are not required to be ordered. The array layout for A_{ex} in COO format is given in Table 3.1 below for a one-based array indexing scheme. The total amount of storage required by COO is $2NNZ$ integers + NNZ reals.

As an example, the following pseudo-code will perform the matrix-vector multiplication $w = Ad$, where A is stored in COO format.

for *i = 1* **to** NNZ **do**

w[i] = 0.0;

endfor

for *i = 1* **to** NNZ **do**

$w[\texttt{row\_inds}[i]] = w[\texttt{row\_inds}[i]] + \texttt{values}[i] * d[\texttt{col\_inds}[i]];$

Table 3.2: CSR Storage Layout for A_{ex} - Blank `row_ptrs` entries are not allocated

Array Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`row_ptrs`	1	5	9	11	13	15	18							
`col_inds`	1	2	3	4	1	2	3	5	2	3	4	6	1	4
`values`	11	12	13	14	21	22	23	25	32	33	44	46	51	54

Array Index	15	16	17
`row_ptrs`			
`col_inds`	3	5	6
`values`	63	65	66

endfor

3.3.1.2 Compressed Sparse Row (CSR) Data Structure. The CSR data structure takes advantage of the fact that there is usually more than one element per row. In the CSR data structure, the `col_inds` and `values` arrays have the same meaning as in the COO data structure, but are required to be given row by row. The COO `row_inds` array is replaced by an array of row pointers, called `row_ptrs` in Table 3.2. The i^{th} entry in the `row_ptrs` array indicates the first index in both the `col_inds` and `values` arrays for that row, together with a fixed entry `row_ptrs`$[n+1] = NNZ + 1$ A variation of the CSR data structure is the compressed sparse column (CSC) data structure; which is the same as CSR, except with the roles of the rows and columns reversed. The total storage required by CSR is $NNZ + n + 1$ integers and NNZ reals.

The following pseudo-code will perform the matrix-vector multiplication $w = Ad$, where A is stored in CSR format.

for *row = 1* **to** n **do**

$w[row] = 0.0$

for $idx =$ `row_inds`$[row]$ **to** `row_inds`$[row+1] - 1$ **do**

$w[row] = w[row] +$ `values`$[idx] * d[$`col_inds`$[idx]]$;

endfor

endfor

3.3.1.3 Modified Sparse Row (MSR) Data Structure. The MSR data structure is used for matrices that will have the diagonal entries frequently accessed. There are only two arrays used in the MSR scheme: `indices` and `values`. The diagonal entries are stored in the first n entries of the `values` array, an unused entry is then stored, and then the remaining off-diagonal entries are stored in row-wise fashion. The first n entries of the `indices` array point to the beginning storage location of each off-diagonal row, the $(n+1)^{st}$ entry contains $NNZ+1$ similar to the `row_ptrs` array in the CSR data structure. The remaining `indices` array entries contain the column index for each corresponding entry in the `values` array. The total storage required by MSR is $(NNZ-D)+n+1$ integers $(NNZ-D)+n+1$ reals, where D is the number of diagonal entries $0 \leq D \leq n$.

The following pseudo-code will perform the matrix-vector multiplication $w = Ad$, where A is stored in MSR format.

for *row = 1* **to** n **do**

Table 3.3: MSR Storage Layout for A_{ex} The "x" value is unused.

Array Index	1	2	3	4	5	6	7	8	9	10	11	12	13	14
`indices`	8	11	14	15	16	18	20	2	3	4	1	3	5	2
`values`	11	22	33	44	0	66	x	12	13	14	21	23	25	32

Array Index	15	16	17	18	19
`indices`	6	1	3	5	6
`values`	46	51	54	63	65

$w[row]$ = `values`$[row]$

for `idx =` `indices`$[row]$ **to** `indices`$[row+1]$ $-$ 1 **do**

$w[row] = w[row] +$ `values`$[idx] * d[$`indices`$[idx]]$;

endfor

endfor

3.3.2 Global Data Structures

Traditionally, the global data structure has been a dense matrix of blocks. To get good cache locality, even on computers with large main memories, large matrices may be divided so each processor owns many block rows or block columns, requiring a sparse block matrix structure. The HyperMatrix data structure takes advantage of global sparsity by storing *blocks* in a COO data structure. Figure 3.2 shows a simple example HyperMatrix. The `Matrix` class in HMPE implements a two-level HyperMatrix. At the global level, a list of matrix blocks are stored in a HyperMatrix. At the local level, each matrix block is in CSR format.

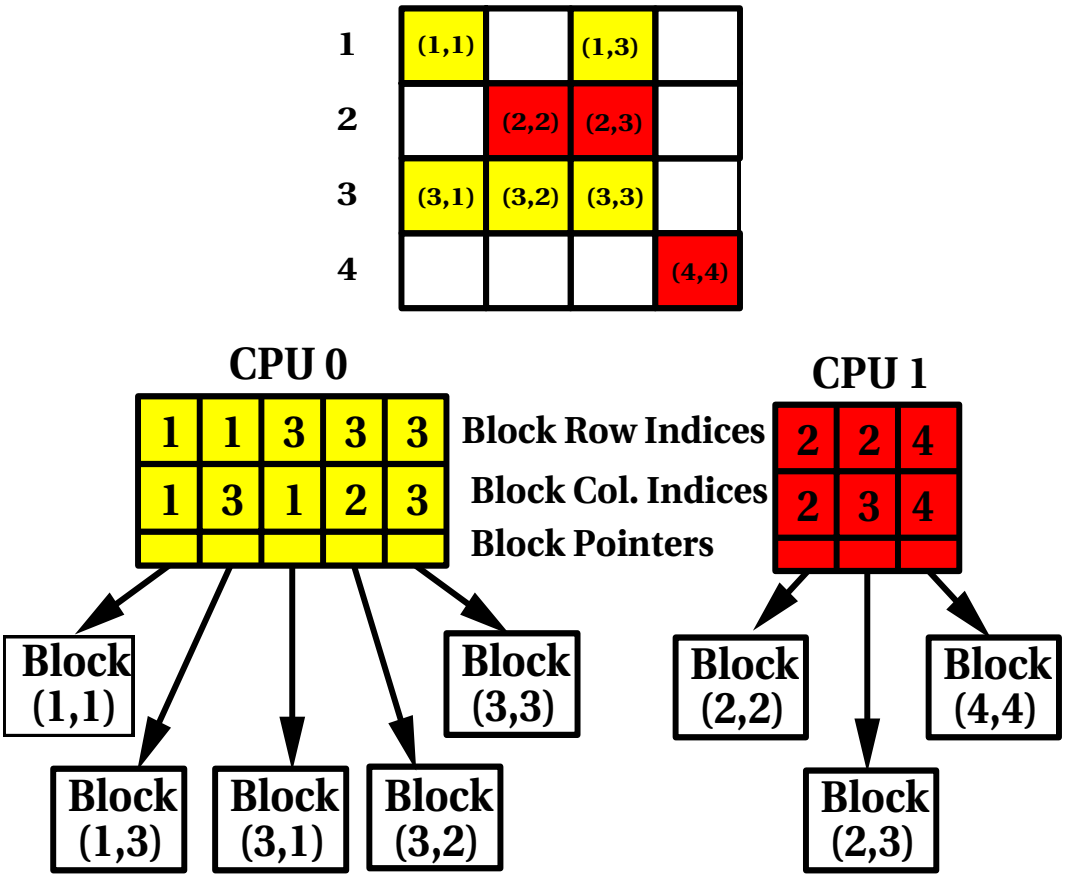

Figure 3.2: HyperMatrix Data Structure Example. The top figure represents a 4×4 block example matrix cyclically distributed among CPUs 0 and 1. The bottom figures represent the corresponding HyperMatrix for each CPU. In HMPE, each block is in CSR format.

3.3.2.1 Proposed HyperMatrix Variations. Other proposals have been made to the basic two-level scheme implemented in HMPE. If each matrix block were allowed to have a different data structure, the block operations can be optimized for the distribution of non-zero entries within the block. Ongoing research is investigating the use of a case-based reasoning system to determine the best per block data structure type given statistics of each matrix block.

As matrices become larger (say $\mathcal{O}(n) = 10^{12}$ and above), the size of computation will exceed that even of the largest parallel computer. An alternative to a single massively parallel system is a shared memory array computer as described in Chapter 2. Shared memory array computers need new matrix data structures to efficiently map sparse systems to the new parallel environment. Allowing the HyperMatrix data blocks to be HyperMatrices (i.e., making the HyperMatrix data structure recursive) would provide a natural mapping to such an architecture. The highest level HyperMatrix would contain one block row or block column for each shared memory processor. Each shared memory processor would have its own HyperMatrix, which could in turn be recursive; communication between shared memory processor would be coordinated through the top level HyperMatrix.

3.4 Design of the HMPE Linear System Solver

The HMPE linear system solver was designed according to the model of a parallel program described in Section 2.3 and serves primarily as a practical test of this model. This section describes the overall design of HMPE, its use of both high-level and low-level kernels, and in lieu of a complete source code listing, an example of high-level kernel implementation is provided.

3.4.1 The HMPE Main Routine

Pseudo-code corresponding to the HMPE main routine is listed below:

Initialize:

start up MPI

read in control file

read in partition file

create distribution D

input matrix A

if *(doing estimation)* **then**

read in estimation files

endif

if *(preconditioning is required)* **then**

factor diagonal blocks of A and store them in M

endif

End Initialization

call the solver

output results

finalize MPI

The majority of the main routine is involved with solver initialization. MPI start up initializes the parallel environment and sets the global variables that indicate the number of processes `(NPROCS)` and this process's ID number `(MYID)`. The control file specifies:

- the solver method
- the global preconditioner and factorization method
- the Krylov subspace size S for GMRES
- matrix input source
- estimation control data
- the maximum number of iterations and residual tolerance ϵ used to determine solver termination

The partition file input determines the number and sizes of each block row of the matrix and correspondingly, the vector block sizes. The distribution assigns each process ownership of one or more block rows/vector blocks in a cyclic fashion based on `NPROCS`. The matrix A is input, either from a file in Harwell-Boeing format [21] or from an internal routine `partsev` that generates finite difference operator matrices, depending on the control parameter listed above. The matrix input routine uses the partition and distribution to input A, one block row at a time by its owning process, and determine communications requirements. If estimation is being performed, the architecture and processor files described in Chapter 2 are read in. If preconditioning is used, the diagonal blocks of A are then factored and stored in M. The solver is called and executed. Upon its return, results are printed, the MPI environment is finalized, and HMPE exits.

3.4.2 Kernels in HMPE

As previously stated, a kernel is a function that performs a basic operation. Examples for linear system solvers are dot products, matrix-vector multiplications, data transmission, and global synchronization. Iterative solver codes can be written as a series of kernel calls along with control statements and some scalar operations. Stating a solver in terms of kernel operations on matrix/vector data structures (implemented

as classes in C++) allows for changing kernel or class implementation while maintaining the solver code [47]. In HMPE, the kernels listed in Appendix A are considered **high-level** kernels which are in turn implemented in terms of other high-level kernels and **low-level** kernels. The high-level kernels provide the solver interface to the low level kernels and perform estimation if necessary. Low-level kernels are generally simple loops or wrappers for library (i.e. BLAS, SPLIB [9], or MPI) routines.

For example, HMPE has evolved from a non-blocked uniprocessor solver to a blocked parallel solver. Originally, the high-level kernels were written as wrappers for calls to underlying kernels, which led to a non-blocked uniprocessor code. Then the high level kernels, along with the `Matrix` and `Vector` classes, were modified to implement a block matrix solver. Finally, parallelism was added – the bulk of the code changes were made to modify the I/O routines and the MATVEC and PRECOND high-level kernels. All the MPI communication kernels used in the parallel version of HMPE use either non-blocking communication or collective communication, primarily reduction operations. None of these transformations required changes to the original solver routines, greatly reducing debugging time from previous attempts.

Expressing solvers in terms of a fixed list of kernels has some drawbacks. Any one list of kernels is likely to be incomplete and nonstandard. Non-kernel code will generally be needed; these special purpose fragments are primarily conditional statements, looping structures, and scalar operations. Combining a series of kernel calls into one

operation or loop may enhance performance by reducing overhead. If the code uses different data structures throughout its execution, kernel usage may require additional overhead for data structure transformations. However, sparse iterative solvers typically are built from the same small set of kernels, with minor variations related to operations on small matrices ($\mathcal{O}(10)$ instead of $\mathcal{O}(n)$). These operations in turn typically require relatively little time.

The advantages of portability, shorter debugging time, and kernel implementation flexibility outweigh the drawbacks. This implementation strategy also allows the assumption that a solver is fundamentally a series of calls to a short, fixed list of kernels. Since the HMPE implementation fits the model of parallel computer performance described in Section 2.3, HMPE provides a test case for the application of that modeling strategy to estimate HMPE's performance.

3.4.3 High-Level Kernel Examples: MATVEC and START_COMM

The MATVEC high-level kernel is an illustrative example because it uses both high-level and low-level kernels and involves communication, by use of a START_COMM kernel. C++ code for the MATVEC and START_COMM high-level kernels is provided in Appendix B.

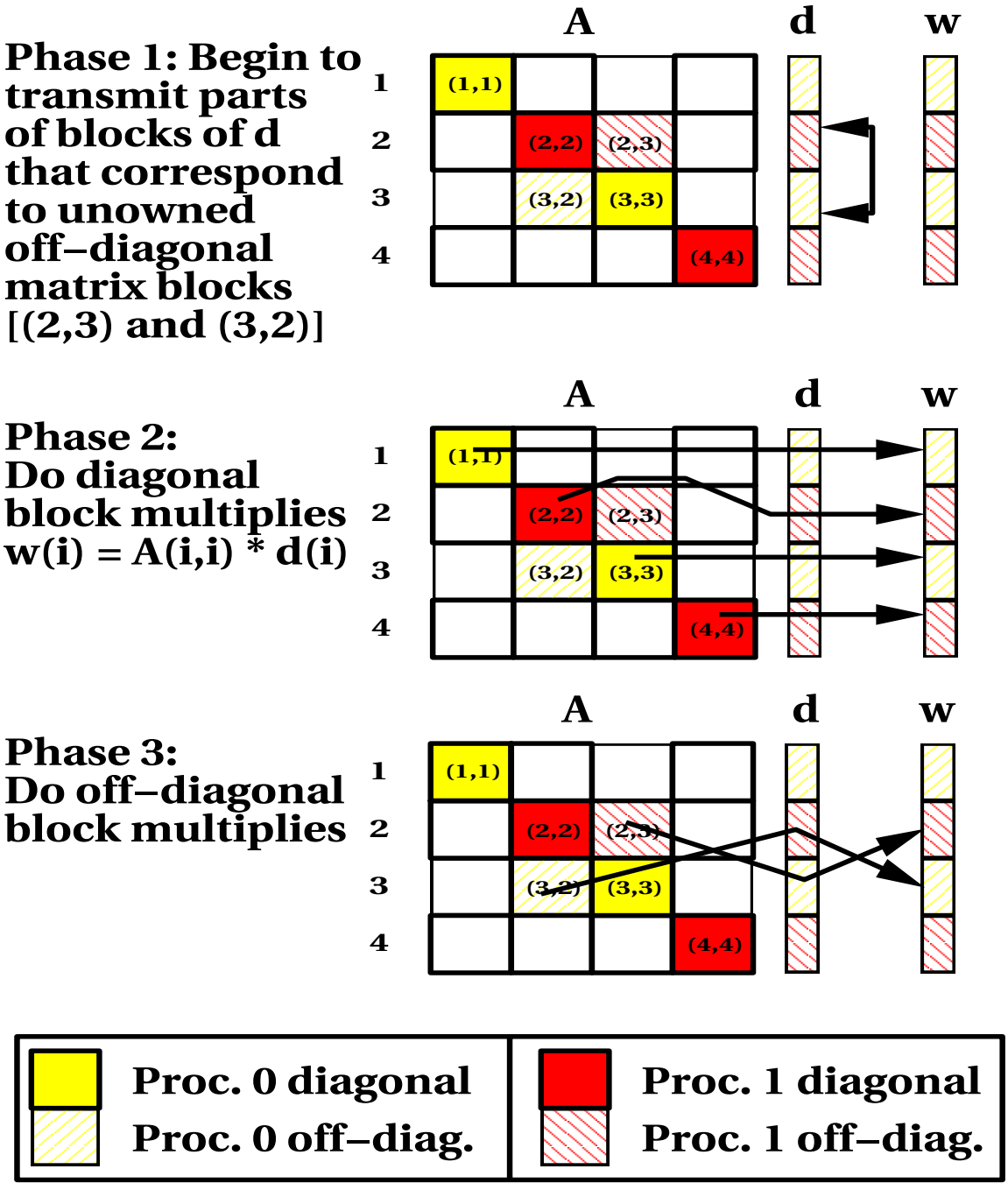

Figure 3.3: HMPE Matrix-Vector Multiplication Example. The operation $w = Ad$ is diagrammed, where A is a 4×4 block matrix cyclically distributed among processes 0 and 1.

MATVEC is comprised of three phases which overlap communications and calculations as much as possible. A simple example is shown in Figure 3.3.

Phase 1. MATVEC starts all required communication, using START_COMM. For each process P, START_COMM initiates communications with all processes that require it; either by calling the MYSEND high-level kernel to start a non-blocking send for vector blocks owned by P or by calling the MYRECV high-level kernel to start a non-blocking receive for vector blocks needed by P.

Phase 2. MATVEC performs the sparse matrix-vector multiplications for the diagonal blocks of A owned by P using the low-level kernel LMATVEC. This is done under the reasonable assumption that the diagonal blocks contain the majority of the non-zero entries of A and requires some time, which may allow the non-blocking communications to complete.

Phase 3. MATVEC loops to check if any communications have been completed. If any new vector blocks are now resident, the corresponding matrix-vector block multiplication with summation is performed using the LMATVECP low-level kernel. When all off-diagonal blocks are accounted for, MATVEC is complete.

3.5 Conclusion and Summary

This chapter presents the basic theory and design behind the HMPE parallel preconditioned iterative solver. HMPE implements high-level kernel operations in terms of three types of low-level kernels: numerical, communications, and synchronization. It was designed to be a practical test of the model of parallel computer performance described in Section 2.3. As will be shown in Chapter 4, this model accurately estimates HMPE's performance.

4

The AET Metric and Computational Results

4.1 Introduction

This chapter describes the AET metric and presents results that show the EC metric does not provide even a qualitative measure of solver run-time while the AET metric provides good estimates of solver run-time, measured in terms of time per solver iteration. Unless otherwise indicated, all results are from HMPE runs made on the SGI Power Challenge described in Chapter 2 and the Krylov subspace size for GMRES $S = 5$.

Section 4.2 provides results that show the impossibility of estimating the number of solver iterations. Section 4.3 describes that the AET metric calculation in terms

of the parallel program model of Section 2.3. Section 4.4 gives results for HMPE and the AET metric – estimates of HMPE performance are always within 13.1% relative error, but are generally within 5% and can be improved. Section 4.5 describes how the computational results to indicate necessary considerations for good solver performance estimation. Finally, Section 4.6 concludes and summarizes the chapter.

4.2 Measuring Solver Execution Time

The iterative solver execution time ET can be calculated as $ET = \text{SIT} + \text{NI} \times \text{TPI}$, where SIT is the solver initialization time, NI is the number of solver iterations and TPI is the time per iteration. Most of SIT is spent in computing a preconditioner, and that time is typically small compared to total solution time, so it is ignored below. As a practical matter, NI cannot be determined ahead of time, so only TPI can be estimated.

It is impossible in general to 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 [3], there are no realistic estimates for practical problems. Three additional complicating factors also arise. First, the targeted systems are nonsymmetric in value. In this case, even a complete *a priori* knowledge of the

Table 4.1: The number of iterations required by CGSTAB for the **sherman3** matrix, for nine different summation orderings in computing the dense dot products.

Ordering Method	1	2	3	4	5	6	7	8	9
Iteration Count	102	109	91	73	97	111	101	103	97

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 is not known. In many cases a preconditioner can actually increase the number of iterations required. Finally, each partitioning implicitly defines a reordering of the matrix, with subsequent changes in the order of operations and quality of preconditioning [19].

Table 4.1[1] shows the number of iterations required by a uniprocessor implementation of CGSTAB for the matrix **sherman3** from the Harwell-Boeing collection of test matrices. Only the order of summation used in computing the dense dot products was varied; the matrix partitioning and matrix-vector products were fixed. Even this simple change causes the number of iterations to vary from 73 to 111. In a parallel environment, the summation ordering problem is further compounded by the unpredictable order of summation between the processors. This affects dot product operations, matrix-vector, and matrix-matrix operations.

[1]Data provided by Etsuko Mizukami.

Table 4.2: The number of iterations required by GMRES with BSSOR preconditioning for the **BFS** matrix, for seven different reorderings of the matrix.

Ordering Method	1	2	3	4	5	6	7
Iteration Count	68	95	83	83	79	98	74

Table 4.2 shows the number of iterations for seven reorderings induced by various partitioning methods of the **BFS** matrix described in Section 4.4 using the GMRES solver and BSSOR preconditioner running with 8 processes. The results confirm the problem of determining the number of iterations due to reordering – by varying only the matrix ordering, the number of iterations ranged between 68 and 95[2].

Since it is impossible to determine the number of solver iterations ahead of time, the AET metric concentrates on the other factor of the total solution time: the time per iteration.

4.3 The AET Metric Calculation

The AET metric is calculated by applying the parallel performance model described in Section 2.3 to HMPE. The model takes a "building block" approach to generating performance estimates, based on simulating a sequence of kernel calls. Each low-level kernel call made by HMPE estimated by a timing model to generate

[2]GMRES with BDIAG preconditioning on **BFS** failed to converge within 500 iterations for all reordering methods, indicating BSSOR's robustness.

$\mathcal{E}_{init}$. The **AET simulator** takes the estimates of HMPE low-level kernels as input and using information about the process state, generates the $\mathcal{E}_{final}$ value. The AET metric value $\mathcal{E}$ is calculated as the sum of the $\mathcal{E}_{final}$ values for one solver iteration to estimate the TPI.

4.3.1 Kernel Modeling

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

```
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);  }
```

There are three types of timing models used to estimate the performance HMPE low-level kernels: a **cached** timing model, a **collective** timing model, and a **piecewise-linear** timing model. The cached timing model implements a slightly modified version of the standard cache cost function [55, p. 38], described in the next paragraph. Cached timing models are used for modeling the majority of kernels whose performance is influenced by memory performance. The collective timing model models binary collective communications algorithms by scaling a cached timing model estimate with a value proportional to the logarithm of the number of processors executing the kernel and is further described in [43]. A general piecewise-linear model is used when the other two models do not apply.

The cached data model estimate is

$$DME(N, S, t_{small}, t_{large}, t_{limit}) = N \cdot t_{est}(N, S, t_{small}, t_{large}, t_{limit}),$$

where N is the data size, S is the processor cache size S, t_{small} is the time per operation on data sets that fit into the processor's cache, t_{large} is the time per operation for data sets that do not fit into the processor's cache, and t_{limit} is used to modify the estimate if the the caching strategy significantly alters the cost per operation.

t_{est} is for N items 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 formula is derived from the standard performance equation [34, p. 57] of a two-level memory hierarchy:

$$t_{est} = \mathcal{H}t_{small} + (1 - \mathcal{H})t_{large} \tag{4.1}$$

where $\mathcal{H}$ is the hit ratio for the cache. For a data item that completely fits in the cache (i.e. $N \leq S$), $\mathcal{H} = 1$ and Formula 4.1 is reduced to $t_{est} = t_{small}$. For an data item that does not fit into cache ($N > S$), the hit ratio $\mathcal{H} = S/N$ and $1 - \mathcal{H} = N/N - S/N = (N - S)/N$. Substituting back into Formula 4.1, $t_{est} = S/Nt_{small} + (N - S)/Nt_{large}$. t_{limit} is used to fix an upper bound on the cost per operation when observed performance does not match the standard formula, leading to $t_{est} = \min(t_{limit}, (St_{small} + (N - S)t_{large})/N)$.

This approximation for the cost of a dot product along with a collective timing model estimate of the reduction cost were summed to generate Figure 4.1. 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

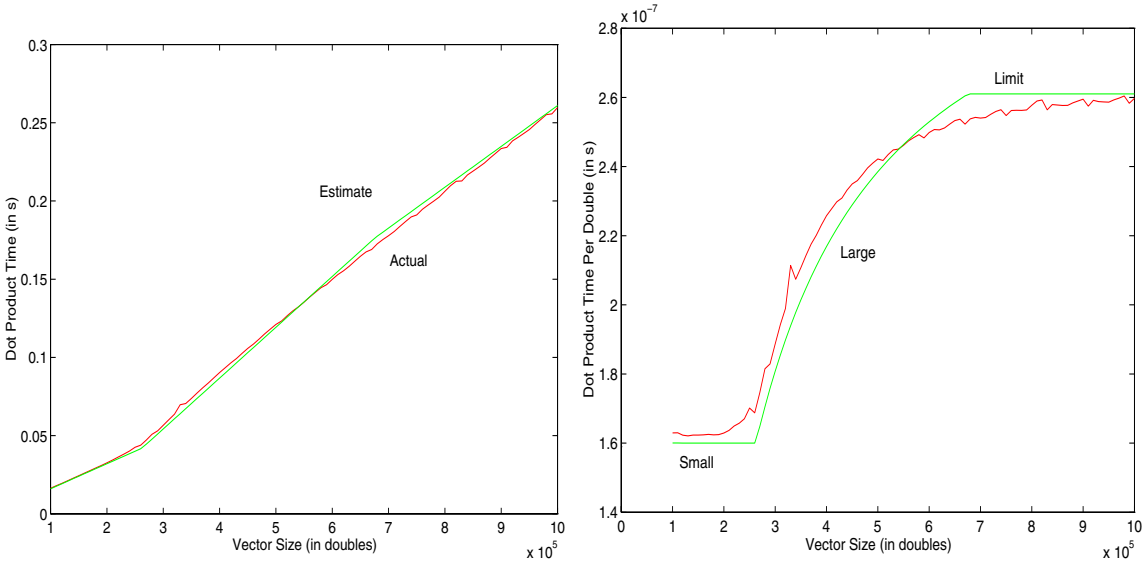

Figure 4.1: Simple kernel modeling. The left hand graph shows the accuracy of the model and the right hand graph shows the effect of the modeling parameters.

per double, which better shows the effect of the t_{small}, t_{large}, and t_{limit} parameters.

4.3.2 The AET Simulator

The AET simulator performs the tasks of the simulator described in Section 2.3.3. The AET simulator runs in parallel as an integral part of each HMPE process. If estimation is required, each high-level HMPE kernel calls the AET simulator to estimate the performance of each low-level kernel call. Each HMPE process also has a process state maintained by the AET simulator.

The process state maintained by the AET simulator is simple: a cache model to provide a two-level memory hierarchy, a process clock to provide $\mathcal{E}$ and for direct

synchronization support, and a barrier bit that indicates if the process is waiting for global synchronization. The cache model implements a LRU (least recently used) cache replacement policy with a correction constant for other policies (i.e. random replacement). Each simulated process state can have a different cache size, scheduling-correction constant, and architecture type. They allow the estimation of processes running on different architectures within the same parallel machine.

The AET simulation code is partially in each high-level kernel and partially in the `MachModel.estimate()` method. Each high-level kernel consists of two parts: actual and estimation kernels. As the names indicate, the actual kernel performs the desired basic operation and the estimation kernel simulates the actual kernel. A pseudo-code version of the whole DOTPROD high-level kernel is given below as an example.

```
double dotprod(Vector A, Vector B) {
    double local, global;

    // actual kernel code
local = 0.0;
    for each block index i of A resident {
        local = local + ldotprod(A.block[i], B.block[i]);
    }
    global = lallsum(local);
```

```
    // estimation kernel code
if (estimating) {
        for each block index i of A resident {
            estimate(DOTPROD, i, A, B);
        }
        estimate(ALLSUM, MYID);
    }
    return(global);
}
```

`estimate()` is shown in the pseudo-code as an overloaded function, but it is implemented as a series of helper functions. These helper functions set up a reference string of vector and matrix blocks used along with their respective sizes, and call the `MachModel.estimate()` method to run the AET simulator for the low-level kernel.

The `MachModel.estimate()` method takes the kernel name, source and destination processes, and reference string as arguments. As setup, the method checks its arguments for validity and determines how many processes are executing the simulated kernel along with each process's architecture type and cache size. After setup, the `MachModel.estimate()` code scans the reference string to update the simulated cache for each block reference and determine the hit ratio estimate $\mathcal{H}_{est}$. Using the

name of the kernel and the sizes of the kernel's inputs, two timing model estimates for the low-level kernel are generated. The first estimate $\mathcal{E}_{cache}$ assumes the data is as fully cached as possible and the second estimate $\mathcal{E}_{uncached}$ assuming fully uncached data. The low-level kernel estimate $\mathcal{E}_{kernel}$ for N items is determined using the standard cached data function: $\mathcal{E}_{kernel} = N * (\mathcal{E}_{cache}\mathcal{H}_{est} + \mathcal{E}_{uncached}(1 - \mathcal{H}_{est}))$. The $\mathcal{E}_{kernel}$ estimates are summed on a per process basis. The process's TPI estimate is then the sum the E_{kernel} values for all kernels executed during one solver iteration. Finally, the AET is the average of all process TPI estimates.

4.4 HMPE Performance Results

The two example matrices are shown in Figures 4.2 and 4.3 by means of the visualization tool EMILY [7]. The matrices arise from computational fluid dynamics problems; the **steady** matrix is from a industrial 3D problem and has order 22 926 with 1 365 036 NNZ and the **BFS** matrix comes from a 2D test problem of order 20 284 and 452 752 NNZ.

Similar speedup curves for HMPE with both matrices are shown in Figure 4.4 for CGSTAB and Figure 4.5 for GMRES. BDIAG exhibits super-linear speedup because the matrix blocks fit into the caches of the individual processors, leading to faster access times. The parallelism inhibiting effects of BSSOR's triangular solves are

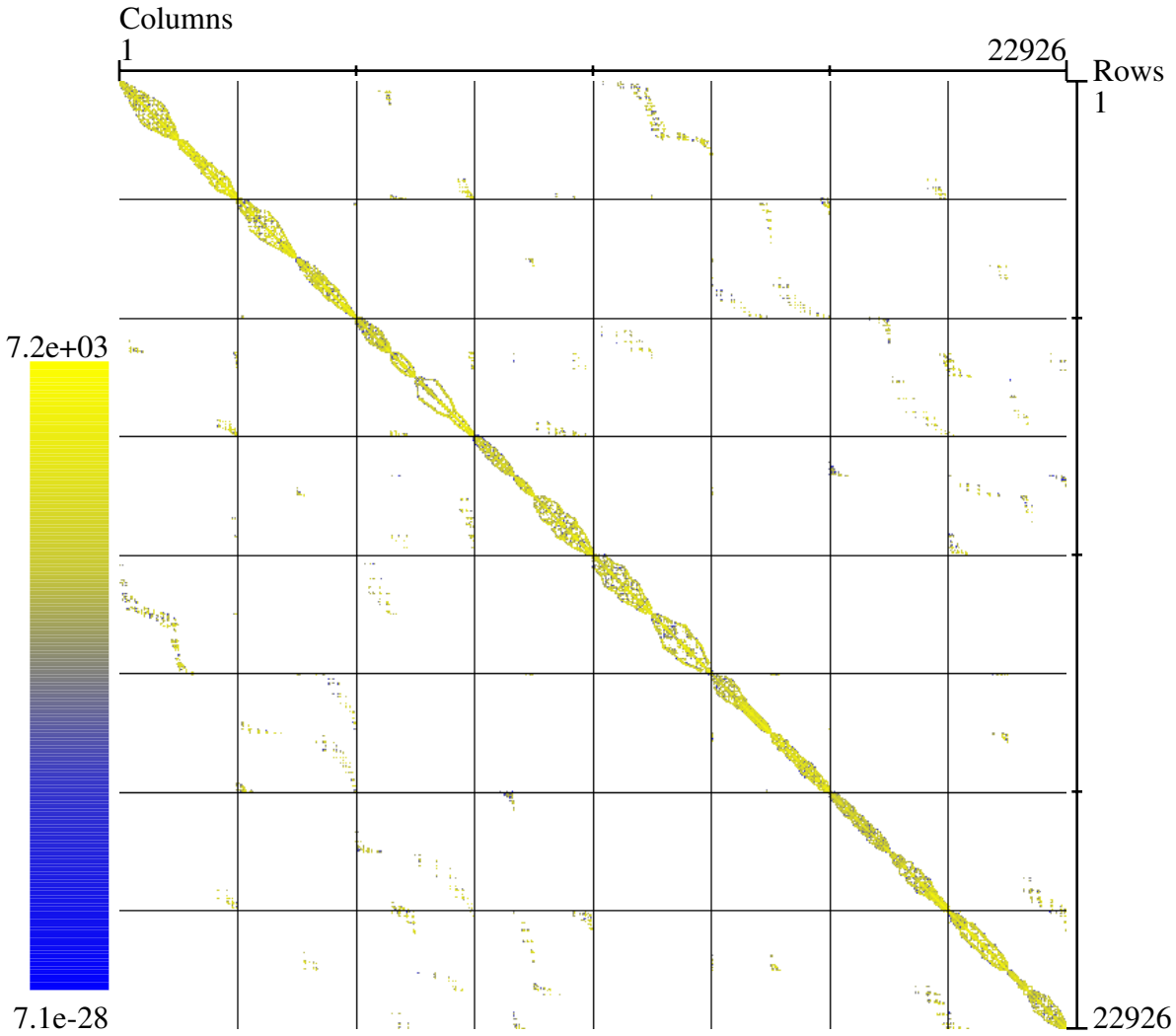

Figure 4.2: EMILY visualization of the **steady** matrix. Both this figure and Figure 4.3 are shown partitioned into 8 block rows using linear partitioning with Robertson's [49] color map.

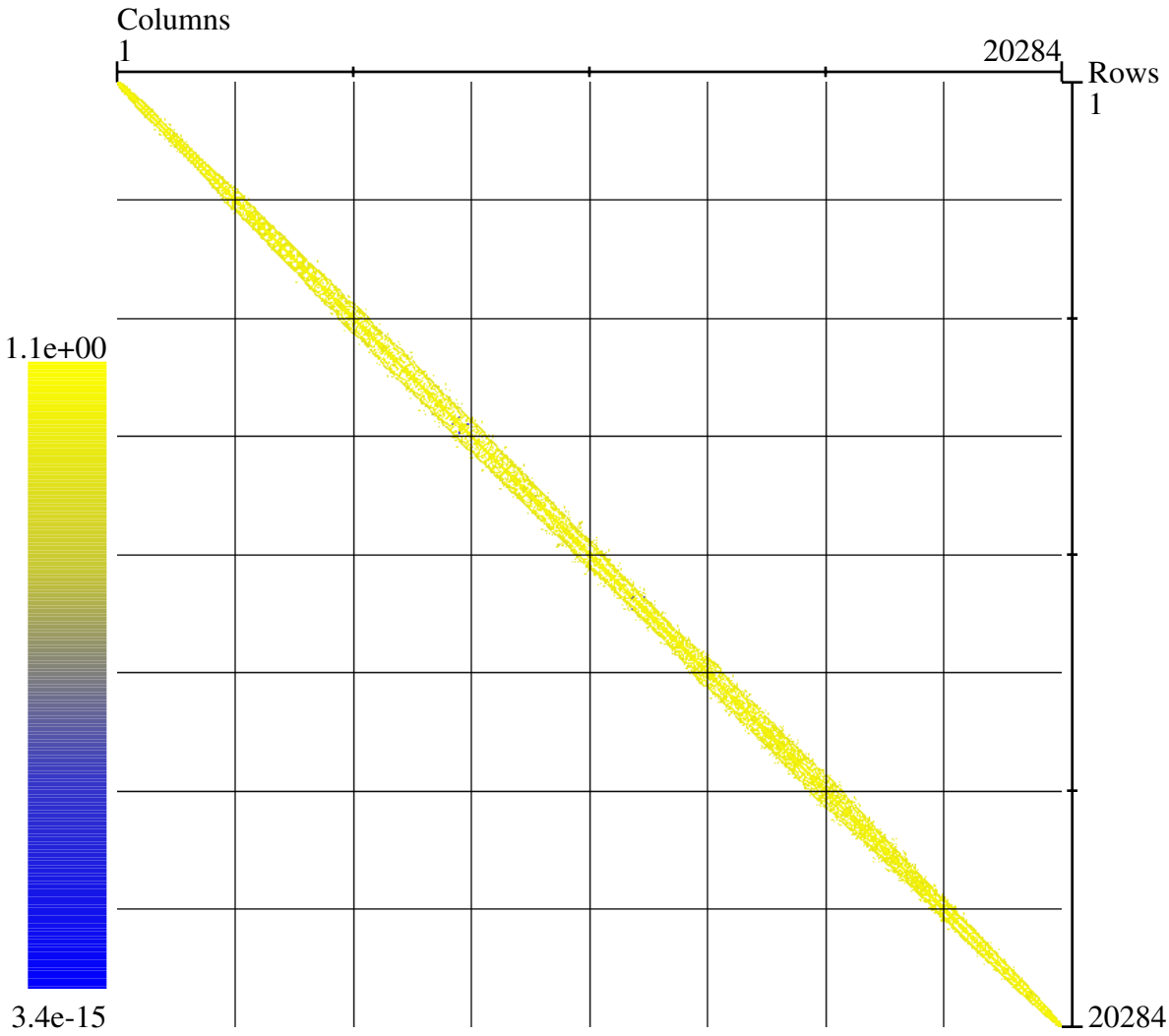

Figure 4.3: EMILY visualization of the **BFS** matrix.

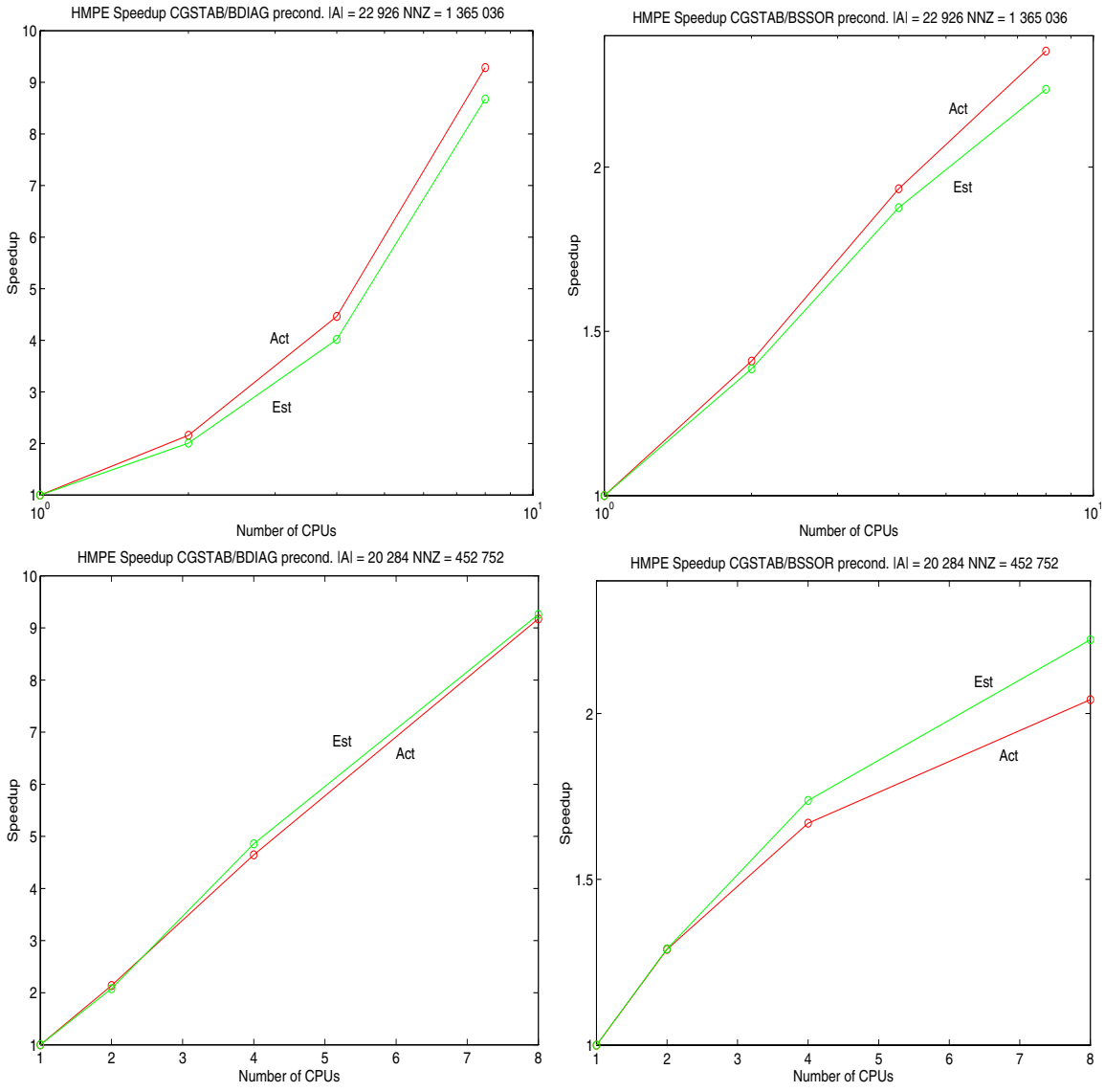

Figure 4.4: Speedup curves for SGI Power Challenge runs using CGSTAB on **steady** (top) and **BFS** (bottom) using BDIAG (left) and BSSOR (right) preconditioning. Super-linear speedup for BDIAG is due to cache effects. Actual results are labeled Act and estimation results are labeled Est.

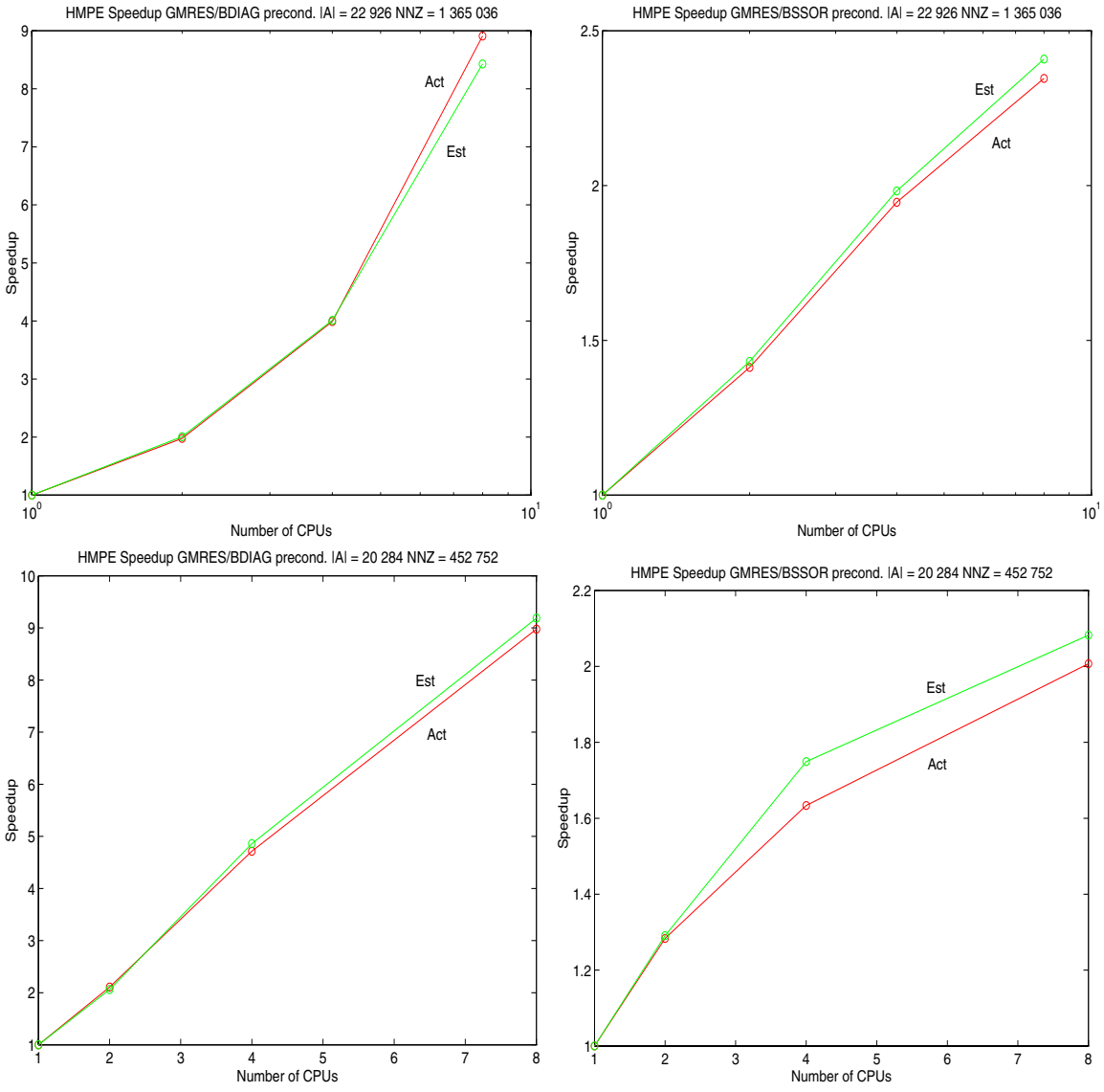

Figure 4.5: Speedup curves for SGI Power Challenge runs using GMRES on **steady** (top) and **BFS** (bottom) using BDIAG (left) and BSSOR (right) preconditioning. Speedup performance is similar to the CGSTAB performance. Actual results are labeled Act and estimation results are labeled Est.

Table 4.3: Partitioning Methods Studied – KL means the local Kernighan-Lin method was used as a post-processing step.

Method	**steady** Edges Cut	**BFS** Edges Cut
LIN	54 772	9 276
LIN-KL	52 229	9 488
MLP	29 468	6 702
Rand-KL	57 773	10 674
SCAT-KL	48 403	9 482
SPECT	43 741	7 672
SPECT-KL	40 979	7 684

evident – BSSOR speedup is between 2 and 2.5 for both matrices using 8 processes compared to about 9 for BDIAG. The AET metric provides good speedup predictions, including BDIAG's super-linear speedup.

4.4.1 The Edges Cut (EC) Metric as a Predictor of the Time Per Iteration (TPI)

To empirically test the number of edges cut (EC) metric as a predictor of the TPI, HMPE was run with CGSTAB and GMRES using the BSSOR and BDIAG block preconditioners. The preconditioning matrix M consisted of the off-diagonal blocks of A and the factored diagonal blocks of A, using incomplete LU factorization with the levels of fill $s = 0$ (ILU(0)). These combinations were chosen as typical of parallel nonsymmetric solvers and use the kernels found in most parallel iterative methods.

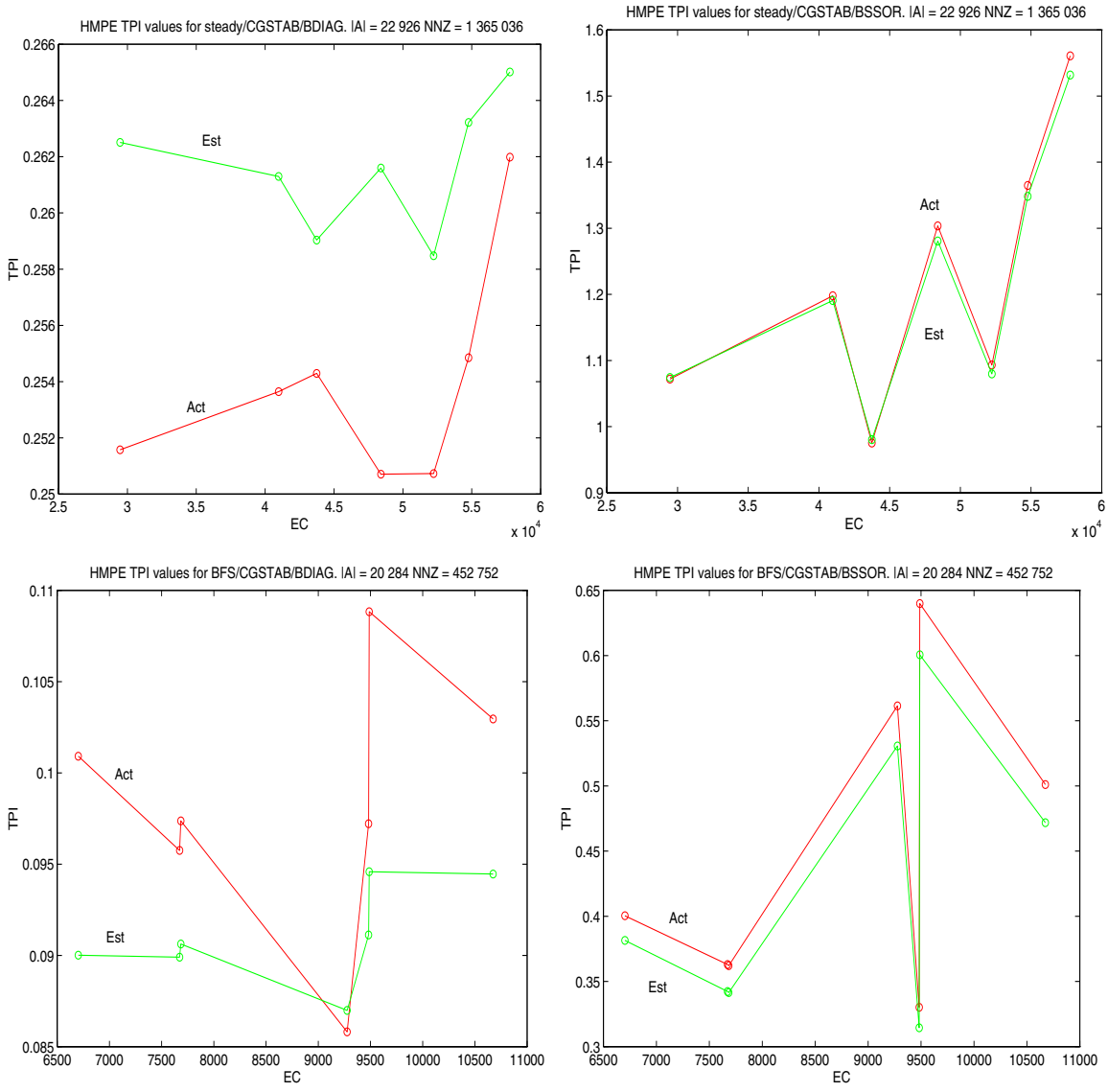

Figure 4.6: CGSTAB Actual and Estimated TPI Results as a function of the number of EC on the SGI Power Challenge. Top left: **steady** with BDIAG preconditioning. Top right: **steady** with BSSOR preconditioning. Bottom left: **BFS** with BDIAG preconditioning. Bottom right: **BFS** with BSSOR preconditioning.

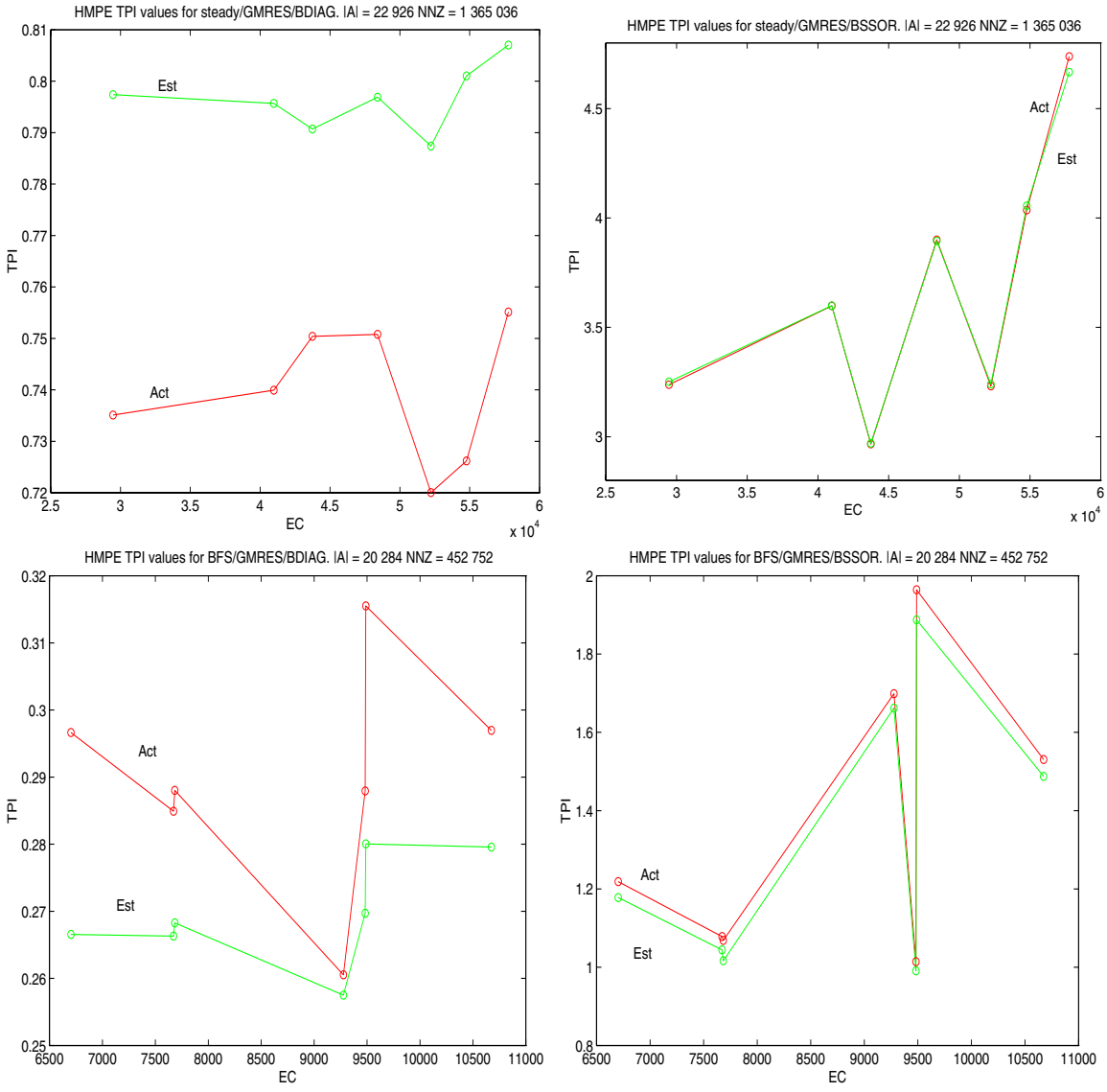

Figure 4.7: GMRES Actual and Estimated TPI as a function of the number of EC on the SGI Power Challenge Top left: **steady** with BDIAG preconditioning. Top right: **steady** with BSSOR preconditioning. Bottom left: **BFS** with BDIAG preconditioning. Bottom right: **BFS** with BSSOR preconditioning.

Seven octa-partitionings of the graphs of the **BFS** and **steady** matrices were generated using Chaco [31] with the methods listed in Table 4.3. The x-axis of each graph of timing in Figure 4.6 shows the number of EC for each partitioning method and the y-axis shows the observed and estimated TPI on the SGI Power Challenge using the CGSTAB solver with ILU(0) preconditioning for each diagonal matrix block. If the number of EC predicts the TPI, each graph in Figure 4.6 should be monotone increasing. However, this clearly is not the case for these examples – as a metric, the number of EC fails to predict the minimum TPI for any of the four matrix/preconditioner pairs. In particular, the lower-right graph of Figure 4.6 shows the results of the BFS matrix using BSSOR preconditioning. The minimum number of EC for all partitionings of BFS is 6 702, yet the minimum TPI (0.326 s) occurs for the partitioning with 9 488 EC, closely followed by the partitionings at 7 672 (0.365 s) and 7 684 EC (0.367 s). The maximum TPI (0.699 s) occurs for the partitioning with 9 488 EC. The difference in the number of EC between the partitionings with the minimum and maximum TPI is 6; yet the ratio of maximum to minimum TPI is 1.95. The number of EC does not even provide a qualitative prediction of the TPI, while the AET BSSOR and BDIAG results have no greater than 8.2% and 13.1% relative error, respectively. GMRES results show similar disparities between EC and TPI. The AET metric predicts GMRES performance somewhat better, likely due to the increased computation required, which leads to more stable timing results.

Table 4.4: CGSTAB Actual and Estimated TPI Data for **BFS** on the SGI Power Challenge

# Procs + Precond	**BFS** Avg t_{act}	**BFS** Avg E (s)	**BFS** Avg R	**BFS** Max R
1+BDIAG	0.8038	0.8254	2.68 %	2.99 %
2+BDIAG	0.3837	0.3999	4.24 %	7.31 %
4+BDIAG	0.1773	0.1720	2.98 %	4.53 %
8+BDIAG	0.0984	0.0911	7.55 %	13.09 %
1+BSSOR	1.186	1.220	2.85 %	3.40 %
2+BSSOR	0.8393	0.8627	2.82 %	3.69 %
4+BSSOR	0.5656	0.5396	4.78 %	6.95 %
8+BSSOR	0.4512	0.4261	5.48 %	6.16 %

The average actual TPI t_{act}, estimated TPI $\mathcal{E}$, and relative error $R = |t_{act} - \mathcal{E}|/t_{act}$, averaged both over all processes and over all partitionings and the maximum relative error for all partitionings were calculated. Tables 4.4 and 4.5 present results for the **bfs** matrix using the CGSTAB and GMRES solvers respectively, and Tables 4.6 and 4.7 present results for the **steady** matrix also using CGSTAB and GMRES respectively.

4.4.2 IBM SP-2 Results and Comparison with Power Challenge

Figure 4.8 shows results from HMPE runs on 8 processors of the IBM SP-2 described in Section 2.1.2 using the **BFS** matrix using CGSTAB with BSSOR preconditioning and ILU(0) factorization. As was seen in the SGI Power Challenge results, AET provides a quantitative estimate of the TPI, whereas the edges cut metric does

Table 4.5: GMRES Actual and Estimated TPI Data for **BFS** on the SGI Power Challenge

# Procs + Precond	**BFS** Avg t_{act} (s)	**BFS** Avg E (s)	**BFS** Avg E	**BFS** Max E
1+BDIAG	2.397	2.425	1.18 %	1.50 %
2+BDIAG	1.144	1.178	2.68 %	4.26 %
4+BDIAG	0.5237	0.5022	4.10 %	5.31 %
8+BDIAG	0.2882	0.2739	5.08 %	9.28 %
1+BSSOR	3.532	3.566	0.97 %	1.45 %
2+BSSOR	2.529	2.521	0.49 %	1.03 %
4+BSSOR	1.693	1.582	6.72 %	8.23 %
8+BSSOR	1.368	1.324	3.23 %	4.90 %

Table 4.6: CGSTAB Actual and Estimated TPI Data for **steady** on the SGI Power Challenge

# Procs + Precond	**steady** Avg t_{act} (s)	**steady** Avg E (s)	**steady** Avg R	**steady** Max R
1+BDIAG	2.210	2.219	2.31 %	6.41 %
2+BDIAG	1.091	1.106	1.29 %	2.41 %
4+BDIAG	0.5381	0.5530	2.77 %	4.01 %
8+BDIAG	0.2540	0.2616	3.01 %	4.34 %
1+BSSOR	3.200	3.328	3.99 %	4.26 %
2+BSSOR	2.312	2.442	5.57 %	6.45 %
4+BSSOR	1.664	1.760	5.76 %	7.43 %
8+BSSOR	1.224	1.212	1.06 %	1.85 %

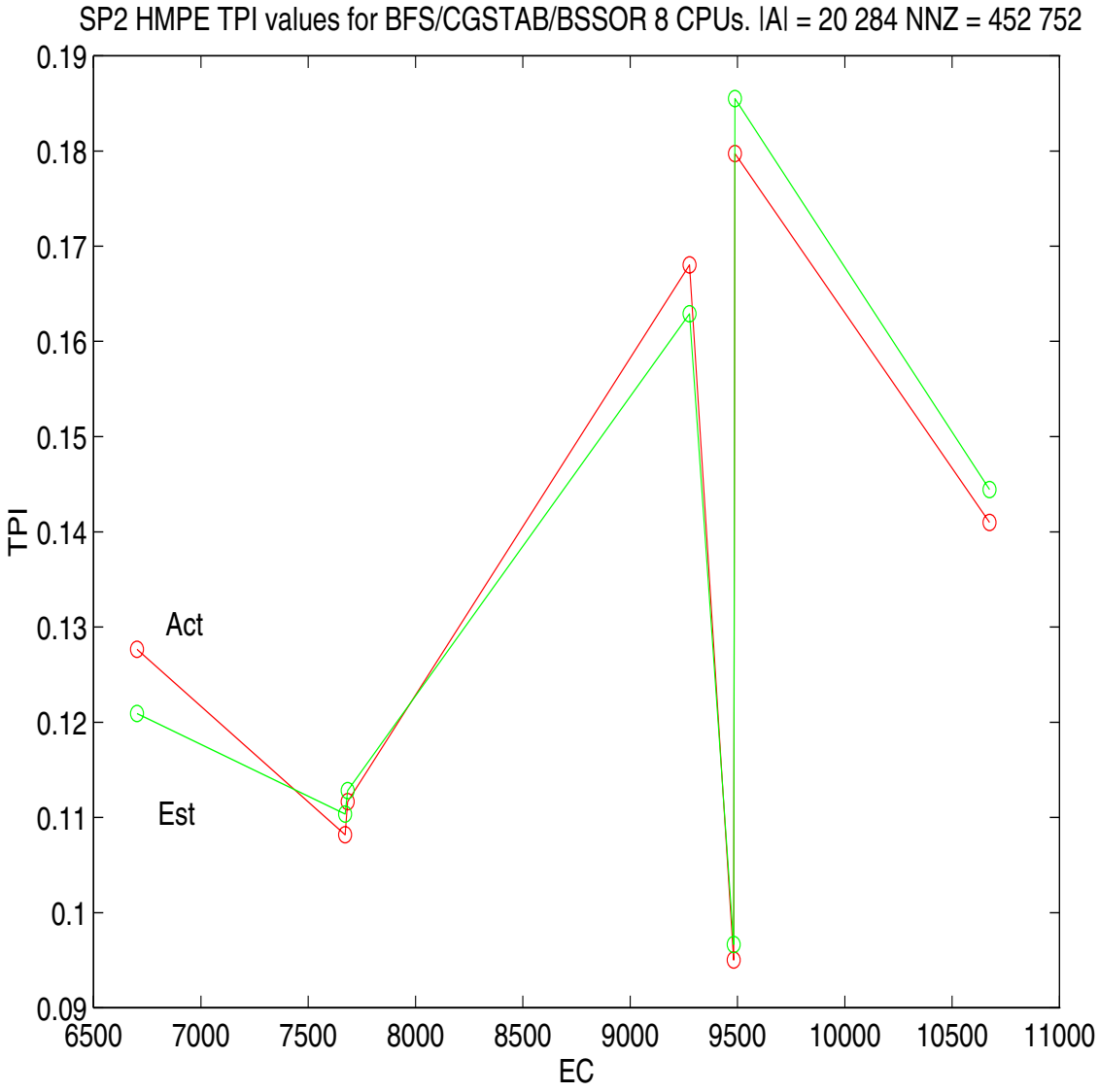

Figure 4.8: IBM SP-2 Actual and Estimated TPI Results on **bfs** with CGSTAB and BSSOR preconditioning

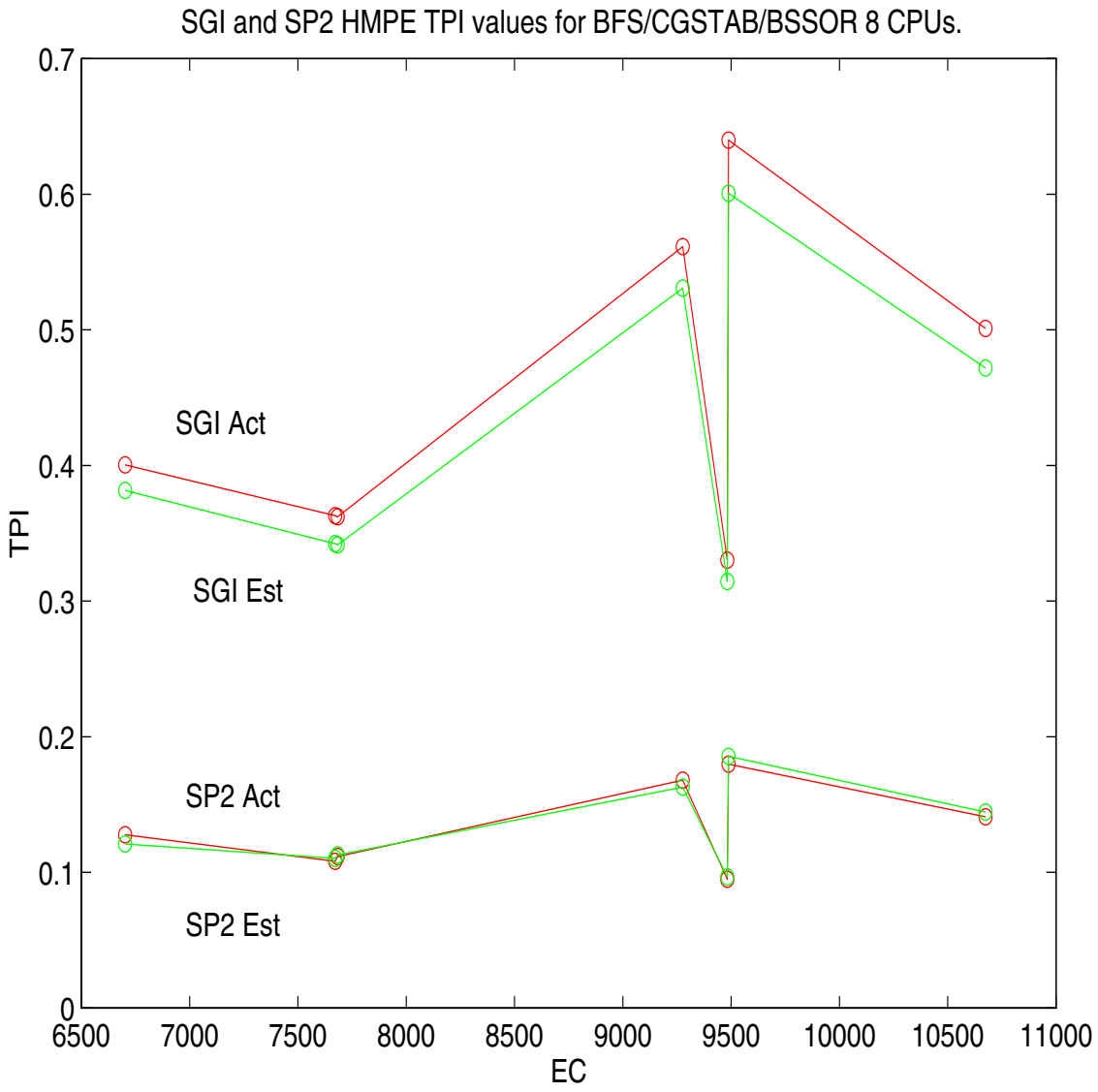

Figure 4.9: IBM SP-2 and SGI Power Challenge Comparison on **BFS** with CGSTAB and BSSOR preconditioning

Table 4.7: GMRES Actual and Estimated TPI Data for **steady** on the SGI Power Challenge

# Procs + Precond	**steady** Avg t_{act} (s)	**steady** Avg E (s)	**steady** Avg R	**steady** Max R
1+BDIAG	6.473	6.528	1.00 %	1.27 %
2+BDIAG	3.278	3.254	0.72 %	1.51 %
4+BDIAG	1.628	1.629	0.85 %	1.86 %
8+BDIAG	0.7396	0.7966	7.72 %	10.31 %
1+BSSOR	9.555	9.792	2.49 %	2.58 %
2+BSSOR	6.931	6.990	0.84 %	1.23 %
4+BSSOR	4.986	4.902	1.76 %	5.86 %
8+BSSOR	3.673	3.668	0.43 %	1.51 %

not.

Figure 4.9 shows a direct comparison of the IBM SP-2 and SGI Power Challenge results for **bfs** with CGSTAB and BSSOR preconditioning. The SP-2 is much faster than the SGI Power Challenge, but the truly remarkable result is the quantitative similarity of the two performance curves. In particular, the ratio of the maximum TPI to the minimum TPI for the IBM SP-2 is 1.89 which is close to the 1.95 value for the SGI Power Challenge. Given the differences between the two machine architecture types, these similarities strongly indicate software and data concerns dominate the quantitative performance of HMPE.

Figure 4.10 shows IBM SP-2 results on the **steady** matrix and Figure 4.11 shows comparison results on **steady** between the SGI Power Challenge and the IBM SP-2. These results further indicate the ability of the AET to predict TPI and the similarity

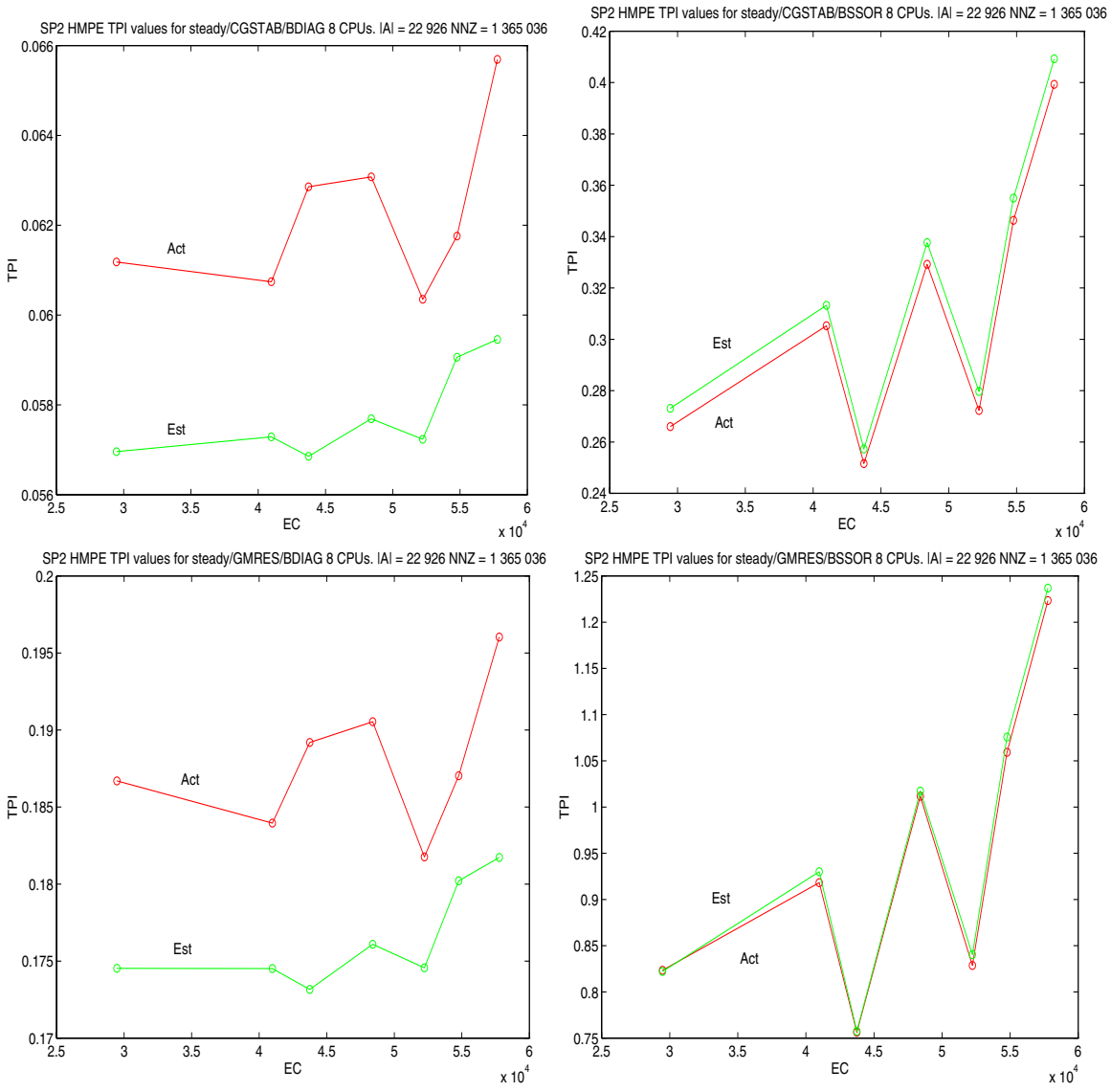

Figure 4.10: IBM SP-2 Actual and Estimated TPI Results as a function of the number of EC on the IBM SP-2. Top left: **steady** with CGSTAB and BDIAG preconditioning. Top right: **steady** with CGSTAB and BSSOR preconditioning. Bottom left: **steady** with CGSTAB and BDIAG preconditioning. Bottom right: **steady** with CGSTAB and BSSOR preconditioning.

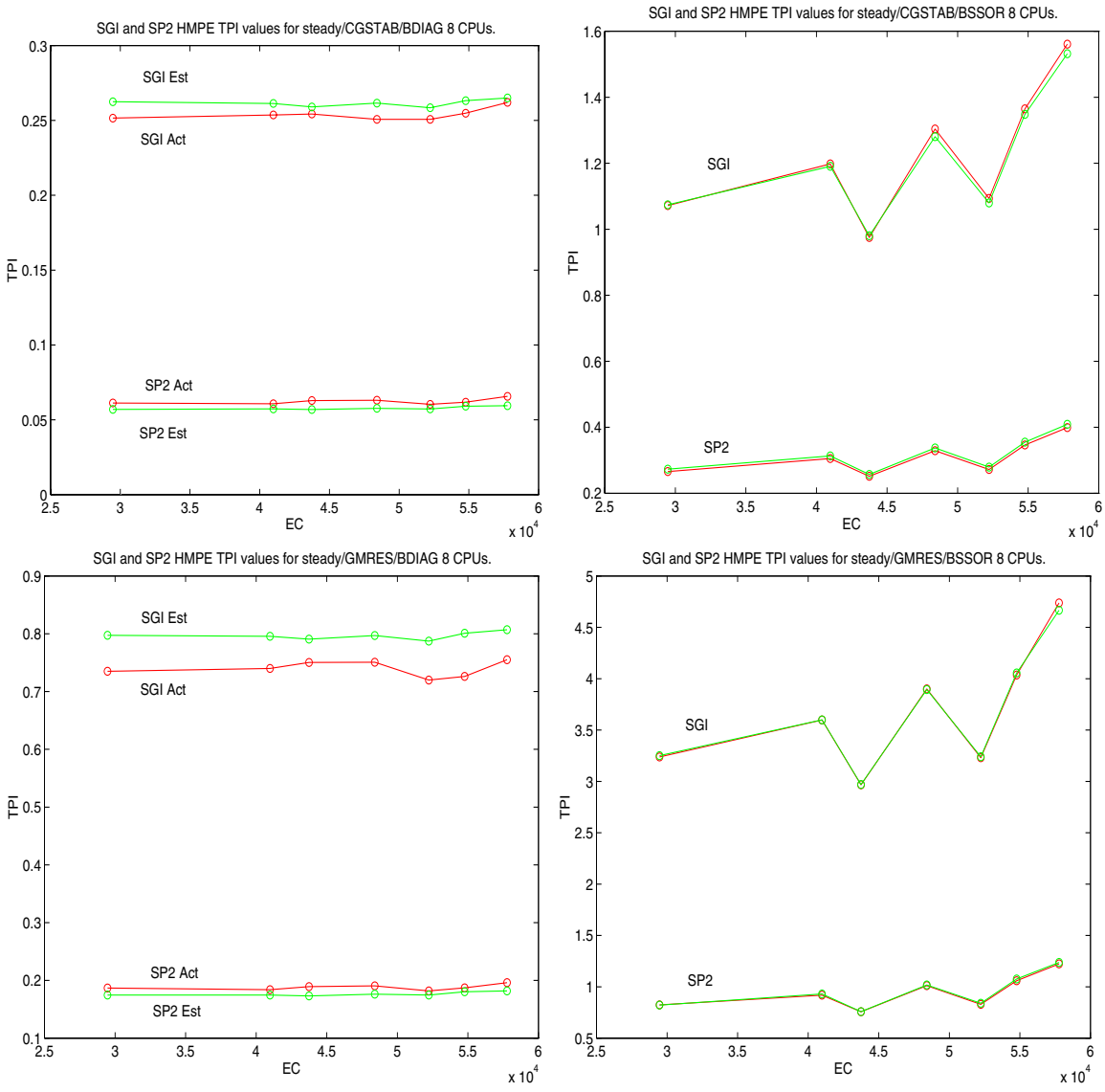

Figure 4.11: Comparison of HMPE Performance on the IBM SP-2 and SGI Power Challenge on the **steady** matrix. Top left: **steady** with CGSTAB and BDIAG preconditioning. Top right: **steady** with CGSTAB and BSSOR preconditioning. Bottom left: **steady** with CGSTAB and BSSOR preconditioning. Bottom right: **steady** with CGSTAB and BDIAG preconditioning.

of quantitative performance of HMPE across the two hardware platforms.

The only code changes made porting HMPE to the IBM SP-2 were to change the calling convention between C++ and FORTRAN and the functions that access low-level timer routines. Both the calling convention and timer accesses are not standard between machines. The data files that contain timing models and the IBM SP-2's cache size were also updated. Deriving the timing model constants takes a large amount of both computer and human time, but was considerably simpler on the IBM SP-2 than on the Power Challenge, since the SGI experience and test codes could be used on the IBM SP-2.

4.5 Considerations for Good Solver Performance Estimation

Figure 4.12 plots the TPI as a function of the number of PROBE kernel calls made during 8 CPUs runs of CGSTAB with BSSOR and BDIAG preconditioning. Recall the PROBE kernel is used to check to see if a non-blocking communications call has completed. As all communications in HMPE are non-blocking, the number of PROBE calls should be directly proportional to the synchronization costs of solver communications. These two figures indicate that CGSTAB with BSSOR costs are proportional to its synchronization cost, while the CGSTAB with BDIAG costs are

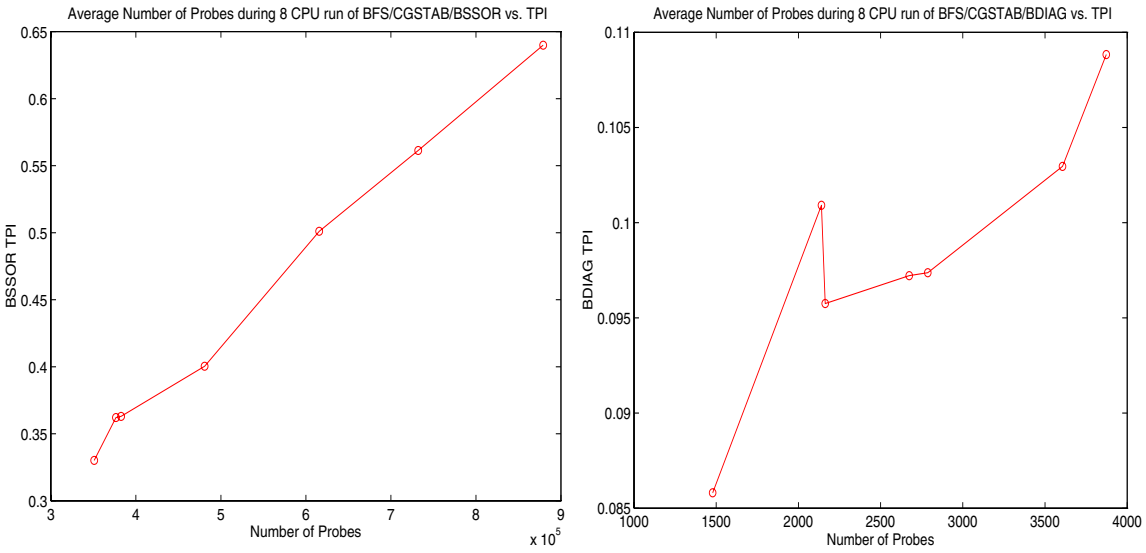

Figure 4.12: TPI of the CGSTAB solver running on 8 processes with BDIAG (left) and BSSOR (right) preconditioning as a function of the number of probe calls made during its run. These results indicate the parallel CGSTAB with BSSOR preconditioning cost proportional to its non-blocking communications synchronization cost, while CGSTAB with BDIAG is not.

not. The CGSTAB with BDIAG plot indicates that other costs, most likely matrix-vector and preconditioner application costs, dominate the more highly parallelizable BDIAG preconditioner.

Preconditioner and solver choices must also be accounted for. The cost of the BSSOR preconditioner clearly dominates the total TPI when more than 2 processes are involved. For example, Table 4.4 shows for 8 processes of CGSTAB with BSSOR on **BFS**, the TPI is 0.4512 seconds vs. 0.0984 seconds for CGSTAB with BDIAG. The matrix clearly must be taken into account; the TPI for CGSTAB with BSSOR on **steady** using 8 processes is 1.368 seconds versus the TPI for CGSTAB with BSSOR

on **BFS** using 8 processes is 0.4512 seconds.

Finally, some model of the computational environment must be made to reflect the performance changes due to the number of processors, cache effects (such as the super-linear BDIAG speedup seen in Figures 4.4 and 4.5), and different architecture types. Our results indicate good TPI estimates need to account for synchronization costs, the choice of solver and preconditioner, the computational environment, and the input matrix, all of which are inputs to the AET metric.

4.6 Summary

The calculation of the AET metric has been described as an implementation of the performance modeling strategy of Section 2.3. AET performance figures compare favorably with the standard graph partitioning accuracy performance metric, the number of edges cut, for predicting solver performance and provides good estimates overall on both the SGI Power Challenge and the IBM SP-2 computers. The comparison results of the two platforms indicate the quantitative performance of HMPE and AET are consistent across architectures.

5

Conclusions and Future Directions

A performance modeling strategy for parallel programs was presented, based on assuming a parallel program's performance is dominated by the time taken by a series of calls to a known list of kernel operations. Using a parallel machine simulation, the model takes into account the cost of each kernel operation and combines them to estimate the program's performance. The HMPE linear system solver was implemented as a practical test of this performance modeling strategy.

Results presented in Table 4.1 indicate that the total number of parallel solver iterations cannot be reliably predicted due to different orders of operations, even if the same solver and matrix are tested. This indicates that the time per iteration (TPI) is the only practical measure of the quality of a data distribution. Analysis of results indicate a good TPI estimator, such as the AET metric, needs to consider synchronization costs, the choice of solver and preconditioner, the computational

environment, and the input matrix. Consideration of these factors allows the AET metric to provide the best currently available TPI estimates.

In general, the number of edges cut (EC) metric does not qualitatively predict the TPI for preconditioned parallel solvers. The edges cut metric has a worse predictive performance for the BSSOR preconditioner, which requires large amounts of synchronization. This provides an indication that metrics such as edges cut which do not take synchronization into account will not provide good TPI estimates.

Future work includes using a more sophisticated process state model, such as described in [2], that could be added into the AET calculation. The AET simulator could be rewritten as separate from the actual solver code using only one simulator process, allowing the use of the AET metric as a cost function in a data distribution algorithm. The use of the parallel computer performance model for run-time estimation in other applications, such as parallel rendering systems, could be investigated.

This dissertation indicates that a bottom-up modeling strategy, which is necessarily tuned a specific application, provides the best currently available predictions of run-time for that application. Using such a "building block" strategy not only provides an estimation method but also a method of easing the task of coding complex applications and porting them from uni-processor to multi-processor environments. These two benefits – the enhanced ability to predict performance and the reduction of coding time in porting a complex application – justify the effort spent in identifying

and coding applications in terms of kernels.

A

List of Kernels Used in Solver Specifications

The following list of high-level kernels are used in the algorithm statements of Appendix B. All but the AXPY2 kernel are implemented in the HMPE solver.

i. AXPY(α, x, y): usual dense saxpy $y = y + \alpha x$

ii. AXPY2(α, x, y): reverse dense saxpy $y = x + \alpha y$. Note: not implemented in HMPE.

iii. DOTPROD(x, y): dense inner product of two vectors $\gamma = x^T w$

iv. VECCOPY(x, y): dense vector copy $y = x$

v. NORM2(x): dense inner product of vector with self $\gamma = x^T x$

vi. MATVEC(A, x, y): sparse symmetric matrix times dense vector $y = Ax$.

vii. VECADD(x, y, z): dense vector add $z = x + y$

viii. VECSUB(x, y, z): dense vector subtract $z = x - y$

ix. UTSOLVE(U, y, x): dense upper triangular solve with dense vector y; $Ux = y$

x. LTSOLVE(L, y, x): dense lower triangular solve with dense vector y; $Lb = b$

xi. VECSCALE(α, x, y): scale a dense vector $y = \alpha x$.

xii. PRECOND(A, M, x, y): apply preconditioner $y = M^{-1}x$. The off-diagonal blocks of M may be stored in A. The block preconditioner choice is assumed to be stored in a global variable.

xiii. GATHER(x, i, b): partial vector copy of x into buffer b using index vector i $b = x[i]$.

xiv. SCATTER(x, i, b): partial vector write of x from buffer b using index vector i $x[i] = b$.

xv. MYSEND$(b, sz, P, req, vnm, done)$: initiate non-blocking send of a buffer b of size sz with name vnm using `MPI_Request` req to process P and set $done$ to indicate completion.

xv. MYRECV$(b, sz, P, req, vnm, done)$: initiate non-blocking receive of a buffer b of size sz with name vnm using `MPI_Request` req to process P and set $done$ to indicate completion.

xvii. MYPROBE$(req, done)$: non-blocking test for message receipt of `MPI_Request` req setting $done$ to indicate message completion.

xvii. START_COMM(A, x): start non-blocking communication of vector b for MATVEC.

xviii. MATVEC_PROBE$(A, x, done)$: check for completion of MATVEC communication.

xvix. BROADCAST_VECTOR_BLOCK$(A, x, bnum)$ broadcast block $bnum$ of vector x.

xx. $\beta =$ALLSUM(α): store in β the result of a parallel reduction of α.

xxi. BARRIER(): global synchronization primitive.

B

Solver and Block Preconditioner Algorithm Statements

All solvers here are specified using pseudo code in terms of the list of kernels given in Appendix A.

All solvers are assumed to have a common calling sequence:

`solver(Matrix` A, `Matrix` M, `Vector` x, `Vector` b,

`int maxits, int K, double` ϵ`)`

where:

- $Ax = b$ is the system to be solved
- M is the preconditioner matrix
- `maxits` is the maximum number of iterations
- `K` is the size of the Krylov subspace for GMRES (unused otherwise)
- ϵ is the maximum residual norm allowed

B.1 The Conjugate Gradient (CG) Algorithm

CG is the one of the oldest iterative methods based on Krylov subspaces; it only works if A and M are symmetric positive definite matrices.

```
CG(Matrix A, Matrix M, Vector x, Vector b,
    int maxits, int K, double ε)
```

begin

```
Vector d, r, w, v;
double j, temp, converged, tol1, rnorm, α, β;
```

Initialize:

```
MATVEC(A, x, r);
VECSUB(b, r, r);
VECCOPY(r, d);
rnorm = NORM2(r);
tol1 = rnorm * ε;
j = 1;
```

End Initialization

while *((rnorm > ε)* **and** *(j <= maxits))*

```
MATVEC(A, d, v);
PRECOND(A, M, v, w);
temp = DOTPROD(d, w);
```

α = rnorm / temp;

AXPY(α, d, x);

AXPY($-\alpha, w, r$);

temp = NORM2(r);

β = temp / rnorm;

rnorm = temp;

AXPY2(β, r, d);

k = k + 1;

endwhile

end

B.2 The Bi-Conjugate Gradient Stabilized (CGSTAB) Algorithm

This statement and the GMRES statement are based on the HMPE code, both of which are in turn based on the SPLIB library [9] code.

CGSTAB(Matrix A, Matrix M, Vector x, Vector b,

int maxits, int K, doubleϵ)

begin

Vector p, q, r, s, t, v, te;

`double j, temp, converged, tol1, rnorm,` $\alpha,\ \beta,\ \hat{\beta},\ \Omega,\ \hat{\Omega},\ \tau$*;*

Initialize:

`tolsq` $= \epsilon \times \epsilon$*;*

MATVEC(A, x, v)*;*

VECSUB(b, v, v)*;*

PRECOND(A, M, v, r)

`rnorm =` NORM2(r)*;*

if `(rnorm <= tolsq)` **then** *return;* **endif**

VECCOPY(r, p)*;*

VECSET$(0.0,\ v)$*;*

VECSET$(0.0,\ q)$*;*

$\alpha = \beta = \hat{\Omega} = 1.0$*;*

`j = 1;`

End Initialization

while *((* `rnorm >` ϵ*)* **and** *(* `j <= maxits`*))*

$\hat{\beta}$ = DOTPROD(p, r)*;*

if *(*$\hat{\beta}$ `is too small`*)* **then** *return;* **endif**

$\Omega = \hat{\beta}\ /\ \beta \times \hat{\Omega}\ /\ \alpha$*;*

$\beta = \hat{\beta}$

AXPY*(-*α, v, q*);*

VECSCALE(Ω, q, q);

VECADD(q, r, q);

MATVEC(A, q, te);

PRECOND(A, M, te, v)

τ = DOTPROD(p, v);

if (τ `is too small`) **then** *return;* **endif**

$\hat{\Omega}$ = $\hat{\beta}$ / τ;

VECSCALE$(\hat{\Omega}, v, s)$;

VECSUB(r, s, s);

MATVEC(A, s, te);

PRECOND(A, M, te, t);

σ = DOTPROD(t, s);

τ = NORM$2(t)$;

α = σ / τ;

if (α `is too small`) **then** *return;* **endif**

AXPY$(\hat{\Omega}, q, s)$;

AXPY(α, s, x);

VECSCALE(α, t, r);

VECSUB(s, r, r);

`rnorm` = NORM$2(r)$;

```
      j = j + 1;
    endwhile
  end
```

B.3 The Generalized Minimum Residual (GMRES) Algorithm

There is a considerable amount of non-kernel code in GMRES, but most of it is either $\mathcal{O}(1)$ or $\mathcal{O}(K)$, where K is the maximum Krylov subspace size. In practice, K is very small compared to n, the order of the matrix.

```
GMRES(Matrix A, Matrix M, Vector x, Vector b,
      int maxits, int K, double ε)
  begin
    Vector v[K+1],r;
    int i, OUTiters, first;
    double H[K+1][K+1], rs[K+1], s[K+1], c[K+1],ro, γ;
    Initialize:
    for i = 1 to (K+1)
      VECSET(v[i],0.0); endfor
    for i = 1 to (K+1)
```

```
   rs[i] = s[i] = c[i] = 0.0;
   for  j = 1 to  (K+1)
     H[i][j] = 0.0; endfor
endfor
first = 1;
OUTiters = 0;
```

End Initialization

do `// outer loop`

MATVEC$(A, x,$ `v[0]`$)$*;*

VECSUB$(b,$ `v[0], v[0]`$)$*;*

$ro =$ NORM2 `(v[0])`*;*

$ro = \sqrt{ro}$*;*

`OUTiters = OUTiters + 1`*;*

if *(*ro `is too small`*)* **then** *return;* **endif**

`t` $= 1/ro$*;*

VECSCALE*(* `t,v[0],v[0]`*);*

`rs[0] =` ro*;*

`i = -1;`

do `// inner loop`

`i = i + 1`*;*

PRECOND($A, M,$ `v[i]`,r);

if *((* `first`*)* **and** *(* NORM2(r) `is too big`*))* **then** *return;* **endif**

`first = 0`;

MATVEC($A, r,$ `v[i+1]`);

for `j = 1` **to** `i + 1`

  `t =` DOTPROD `(v[j],v[i+1])`;

  `H[j][i] = t`;

  AXPY`(-t,v[j],v[i+1])`;

endfor

`t =` NORM2 `v[i+1]`; `t =` $\sqrt{\texttt{t}}$;

`H[i][i+1] = t`;

if *(* `t is too small`*)* **then** *return;* **endif**

for `k = 2` **to** `i`

  `t = H[k-1][i]`;

  `H[k-1][i] = c[k-1] * t + s[k-1]* H[k][i]`;

  `H[k][i] = -s[k-1] * t + c[k-1]* H[k][i]`;

endfor

γ = $\sqrt{\texttt{H[i][i]} * * 2 + \texttt{H [i][i} + 1] * * 2}$;

if *(*γ `is too small`*)* **then** *return;* **endif**

`c[i] = H[i][i]`/ γ;

```
  s[i] = H[i][i+1]/ γ;
  rs[i+1] = -s[i] * rs[i];
  rs[i] = -s[i] * rs[i];
  H[i][i] = c[i]*H[i][i] + s[i]*H[i][i+1];
  ro = | rs[i + 1]|;
while ( (i < (K-1) and (ro ≥ ϵ));
UTSOLVE (H,rs,rs);
VECSCALE (rs[0],v[0],r);
for  j = 1 to  i  AXPY( rs[j],v[j],r); endfor
PRECOND(A, M, r, r);
VECADD(r, x, x);
for  j = 1 to  i
   jj = i1-j;
   rs[jj-1] = s[jj-1]*rs[jj];
   rs[jj] = c[jj-1]*rs[jj];
endfor
for  j = 1 to  i+1
   t = rs[j];
   if (j == 1) then  t = t - 1.0; endif
   AXPY(t,v[j],v[0]);
```

endfor

while *(* `(OUTiters < maxits)` **and** $(ro \geq \epsilon)$*);*

end

B.4 The Block Diagonal (BDIAG) Preconditioning Algorithm

Both the BDIAG and BSSOR block algorithms are stated in terms of a mixture of high and low-level kernels. The fundamental low-level kernel is LPRECOND, which performs both lower and upper triangular solve on the (partially) factored matrix block $M[b]$. The M preconditioning matrix only stores the factored diagonal blocks of A. Note that the LPRECOND low-level kernel performs both a lower and an upper triangular solve using $M[b]$

The i^{th} diagonal block of M is written as $M[i]$ and the corresponding block of vector x is $x[i]$.

`BDIAG(Matrix` A, `Matrix` M, `Vector` x, `Vector` y`)`

begin

`int b;`

for `b = 1 to M.num_block_rows`

LPRECOND$(M[b], x[b], y[b])$*;*

endfor

end

B.5 Block SSOR (BSSOR) Preconditioning Algorithm

The BSSOR algorithm requires access to both diagonal and off-diagonal blocks of A. The off-diagonal blocks are not generally important enough to justfify factorization, so the unfactored off-diagonal blocks of A are used along with the factored diagonal blocks, stored in M. Kernels whose names start with L are low-level kernels.

```
BSSOR(Matrix A, Matrix M, Vector x, Vector y)
begin
  VECCOPY(x, y);
  VECSET(0.0, t);
  for k = 1 to A.num_block_rows
    for i = 1 to A.num_blocks
      row = A[i].row_coord;
        col = A[i].col_coord;
        if ((col == (k-1)) and (row >= k)) then
          LMATVECP(A[i], y[col], t[row]);
```

```
        endif
    endfor
    if (this process owns block row k) then
        LVECSUB(y[k], t[k], y[k]);
        LPRECOND(M[M.diagidx[k]], y[k], y[k]);
    endif
    broadcast_vector_block(A, y, k);
endfor
VECSET(0.0, t);
z = A.num_block_rows;
LVECCOPY(y[z], t[z]);
for  k = A.num_block_rows - 1 downto 1
    for  i = 1 to  A.num_blocks
        row = A[i].row_coord;
        col = A[i].col_coord;
        if  ((col == (k+1)) and  (row <= k)) then
            LMATVECP(A[i], t[col], t[row]);
        endif
    endfor
    if (this process owns block row k) then
```

```
        LPRECOND(M[M.diagidx[k]], t[k], t[k]);
        LVECSUB(y[k], t[k], t[k]);
      endif
      broadcast_vector_block(t, k);
    endfor
    VECCOPY(t, y);
  end
```

The `broadcast_vector_block(v,k)` routine broadcasts block `k` of a vector `v` to all processes who need part of it.

```
broadcast_vector_block(Vector v, int k)
 begin
   owner = v.blocks[k].owner;
   if ( this process != owner) then
     if ( v.recv_sizes[owner][k] > 0) then
       RECV (buff, v.recv_sizes[owner][k], owner);
       while ( not received)
         PROBE();
       endwhile
       SCATTER (v,v[k].recv_indices,buff);
     endif
```

```
  else  // this process owns block k
    for  i = 1 to number of processes
      if ( v.send_sizes[b][i] > 0) then
        GATHER (v, v[k].send_indices[i],buff);
        SEND (buff, v.send_size[i][k], i);
      endif
    endfor
  endif
end
```

B.6 The `Matrix` class and MATVEC Kernel Operation

HMPE's MATVEC implementation in C++ is presented below as an example of a high level kernel's implementation in terms of low level kernels. MATVEC works in three phases: initiation of data communication, which is handled by START_COMM() and is also presented below, diagonal block multiplication, and off-diagonal block multiplication.

The `Matrix` class holds both matrix data and the necessary buffers and communication status variables needed for data transmission.

```
class Matrix

{

public:

    char name[VARNAME_LEN+1];


    int nrows;              // total number of rows

    int ncols;              // total number of columns

    int nnz;                // total number of non-zeros


    // copies of pointers

    int *bstart;            // block start array

    int *blen;              // block length array

    Dist *dist;             // block row ownership


    // local values

    int num_block_rows;     // total number of block rows

    int my_num_block_rows; // this proc's number of block rows

    int cur_info;           // total number of blocks

    int cur_block;          // this proc's number of blocks

    MBlockInfo *info;       // info about ALL blocks
```

```
MBlock *blocks;        // this proc's array of blocks
int *diag_blocks;      // indices of diagonal blocks

int *send_size;        // array of send sizes idx by dest. proc.
double **send_buff;    // send buffers
int **send_sched;      // send schedules for each dest proc...
                       //   see bitmap.C for more details

int *recv_size;        // same structures as used for send
int **recv_sched;
double **recv_buff;

int *comm_done;        // 0 - in prog, 1 - block here
int *mpy_done;         // 0 - block not mulitplied, 1 - block done
MPI_Request *send_req; // send request for each dest.
MPI_Request *recv_req; // recv request for each dest.

Matrix();
~Matrix();
```

```
void init(int zap_ptrs); // initialize non dynamic data

void dealloc();          // return dynamic data to the void


void set_name(char *nm); // set the name of the matrix


// get a copy of global pointers

void set_ptrs(int in_nbr, int *in_bstart,

              int *in_blen, Dist *in_dist);


// set big matrix values

void set_mat_vals(int in_nrows, int in_ncols, int in_nnz);


// allocate dynamic data

void alloc();


// create the block matrix

void build(char *nm, int in_nbr, int *in_bstart, int *in_blen,

   Dist *in_dist, int in_nrows, int in_ncols, int in_nnz);


// add block taken care of in MatrixInput
```

```
    void calc_diag_blocks();

    // communication schedule creation
    void create_comm_scheds(int size, int **send_bitmap,
                            int *recv_bitmap);
};
```

The MATVEC high-level kernel is implemented in terms of the START_COMM and MATVEC_PROBE high-level kernels, and the LMATVEC and LMATVECP low-level kernels.

```
void matvec( Matrix &A, Vector &x, Vector &b )
{
    double t1, t2, tottime;
    int col, row, block;
    int owner, i;
    int nrows, nnz, blocked;
    int done_yet;
    int blocks_done;
    int proc = MYID;
```

```
// reset finished arrays
    for (i = 0; i < A.cur_block; i++)
        A.mpy_done[i] = 0;
    for (i = 0; i < NPROCS; i++) {
        if (i == proc)
            A.comm_done[i] = 1;
        else
            A.comm_done[i] = 0;
    }

// PHASE 1: start off-diag vector transmission
    start_comm(A, x); // start all communication requests

// PHASE 2: for each block row I own, do diag blocks
    i = -1;
    while ((i = x.dist->FindNext(proc,i)) >= 0)
    {
       // get indices
       block = A.diag_blocks[i];
       col = A.blocks[block].info->col_coord;
```

```
        row = A.blocks[block].info->row_coord;

        // do block matvec and keep track
        lmatvec(A.blocks[block], x.blocks[col], b.blocks[row]);
        A.mpy_done[block] = 1;
    }
    // check the off-diag xmission
    done_yet = 0;

// PHASE 3: do off diag block multiplications
    while (!done_yet)
    {
      // look to see if received data
      matvec_probe(A,x,&done_yet); // probe for all outstanding recvs.
                                   // if data present, scatter into x.
      // for each block
      blocks_done = 0;
      for (block = 0; block < A.cur_block; block++) {

          // if this block is done, carry on
```

```
      if (A.mpy_done[block]) {
          blocks_done++;
          continue; }

      col = A.blocks[block].info->col_coord;
      row = A.blocks[block].info->row_coord;

      // get owner of vector corresponding to off-diag block
      owner = A.dist->dist[col];

      // if data from that block is now resident
      if ((owner == proc) || (A.comm_done[owner])) {
          // sum up off-diag block multiplication into b's block
          lmatvecp(A.blocks[block], x.blocks[col], b.blocks[row]);

          A.mpy_done[block] = 1; // mark block done
          blocks_done++;
      }
  } // end of for block loop
} // end of while loop
```

```
}
```

The START_COMM routine is implemented in terms of the GATHER, SCATTER, MYSEND, and MYRECV high-level kernels.

```
//
// communication initiation function.
//
void start_comm(Matrix &A, Vector &x)
{
    int proc = MYID;
    int done;
    int debug = 0;

    int act_send_size, act_recv_size;
    int i;

// quick check
    if (NPROCS <= 1)
        return;
```

```
// for each process -- start off sends
    for (i = 0; i < NPROCS; i++) {

// if something to send, then...
        if (A.send_size[i] > 0) {

            // calc. send size and get the data
            act_send_size = A.send_size[i] - A.num_block_rows;
            gather(x, A.send_sched[i], A.send_buff[i]);

            // start shipping data -- mysend posts the send, too.
            mysend(A.send_buff[i], act_send_size, i, &(A.send_req[i]),
                   x.name, &done);
        }

// if something to receive...
        if (A.recv_size[i] > 0) {
            A.comm_done[i] = 0; // just in case
```

```
            // calc. recv size
            act_recv_size = A.recv_size[i] - A.num_block_rows;

            // start the receive and post it
            myrecv(A.recv_buff[i],act_recv_size, i, &(A.recv_req[i]),
                   x.name, &(A.comm_done[i]));

            // handle case comm is already done.
            if (A.comm_done[i]) {
               scatter(x, A.recv_sched[i], A.recv_buff[i]);
            }
        }
// else done already
        else
        {
            A.comm_done[i] = 1;
        }
    }
// end of for each block loop
}
```

Bibliography

[1] A. Geist, et al, *PVM: A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press, Boston, MA, first ed., 1994.

[2] V. Adve, *Analyzing the Behavior and Performance of Parallel Programs*, PhD thesis, Department of Computer Sciences, University of Wisconsin-Madison, October 1993.

[3] O. Axelsson and G. Lindskog, *On the Eigenvalue Distribution of a Class of Preconditioning Methods*, Numer. Math., 48 (1986), pp. 479–498.

[4] S. Barnard and H. Simon, *A Fast Multilevel Implementation of Recursive Spectral Bisection for Partitioning Unstructured Problems*, in Concurrency: Practice and Experience, also in The Sixth SIAM Conference on Parallel Processing for Scientific Computing, April 1994, pp. 101–117.

[5] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for*

the Solution of Linear Systems: Buildings Blocks for Iterative Methods, SIAM, Philadelphia, PA, first ed., 1994.

[6] R. Blau, *Performance Evaluation for Computer Image Synthesis Systems*, PhD thesis, Department of Computer Science, University of California - Berkeley, December 1992. Available as UCB Tech. Rep. CSD-93-736.

[7] R. Bramley and T. Loos, *EMILY: A Visualization Tool for Large Sparse Matrices*, Tech. Rep. TR 412A, Indiana University Computer Science Department, July 1994.

[8] R. Bramley and V. M. nkov, *Low Rank Off-Diagonal Block Preconditioners for Solving Sparse Linear Systems on Parallel Computers*, Tech. Rep. TR 446, Indiana University Computer Science Department, January 1996.

[9] R. Bramley and X. Wang, *SPLIB: A Library of Iterative Methods for Sparse Linear Systems*, Tech. Rep. 454, Indiana University–Bloomington, Bloomington, IN 47405, 1995.

[10] T. Bui, C. Heigham, C. Jones, and T. Leighton, *Improving the Performance of the Kernighan-Lin and Simulated Annealing Graph Bisection Algorithms*, in Proceedings of the 26th ACM/IEE Design Automation Conference, 1989, pp. 775–781.

[11] T. N. Bui and C. Jones, *Finding Good Approximate Vertex and Edge Partitions is NP-Hard*, Information Processing Letters, 42 (1992), pp. 153–159.

[12] ——, *A Heuristic for Reducing Fill In Sparse Matrix Factorization*, Proceedings of the Sixth SIAM Conference on Parallel Processing, (1993), pp. 445–452.

[13] C. Ashcraft and J.W.H. Liu, *Using Domain Decomposition to Find Graph Bisectors*, Tech. Rep. CS-95-08, York University, November 1995.

[14] Center for Innovative Computer Applications, Indiana University, *A Guide to the Paragon XP/S-A7 Supercomputer at Indiana Univerity*. http://www.cica.indiana.edu/iu_hpc/paragon/paragon.text.html (Visited June, 1996), October 1994.

[15] P. Cousot, *Abstract Interpretation*, ACM Computing Surveys, 28 (1996), pp. 324–328.

[16] D. E. Culler et al, *LogP: A Practical Model of Parallel Computation*, Communications of the ACM, (1996), pp. 78–85.

[17] E. D. Dahl, *Mapping and Compiled Communication on the Connection Machine System*, in Proceedings of the Fifth Distributed Memory Computer Conference, D. W. Walker and Q. F. Stout, eds., Los Alamitos, CA, April 1990, IEEE Computer Society Press.

[18] J. J. Dongarra, S. W. Otto, M. Snir, and D. Walker, *An Introduction to the MPI Standard*, Tech. Rep. http://www.osc.edu/lam.html (Visited October, 1996), The University of Tennessee-Knoxville, January 1995.

[19] I. Duff and G. Meurant, *The Effect of Ordering on Preconditioned Conjugate Gradients*, BIT, 29 (1989), pp. 635–657.

[20] I. S. Duff, A. M. Erisman, and J. K. Reid, *Direct Methods For Sparse Matrices*, Oxford University Press, New York, NY, first ed., 1986.

[21] I. S. Duff, R. G. Grimes, and J. G. Lewis, *Users Guide for the Harwell-Boeing Sparse Matrix Collection (Release I)*, Tech. Rep. TR/PA/92/86, Cedex and Boeing Computer Services, October 1992.

[22] L. C. Dutto, W. G. Habashi, and M. Fortin, *Parallelizable Block Diagonal Preconditioners for the Compressible Navier-Stokes Equations*, Computer Methods in Applied Mechanics and Engineering, 117 (1994), pp. 15–47.

[23] E. Rothberg, *Exploring the Tradeoff Between Imbalance and Separator Size in Nested Dissection Ordering*, Tech. Rep. http://reality.sgi.com/employees/rothberg_asd/ndimbal.ps (Visited October, 1996), Silicon Graphics, Inc., January 1996.

[24] V. Faber and T. Manteuffel, *Necessary and Sufficient Conditions for the Existence of a Conjugate Gradient Method*, SIAM Journal on Numerical Analysis, 21 (1984), pp. 352–361.

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

[26] S. W. Hammond, *Mapping Unstructured Grid Computations to Massively Parallel Computers*, PhD thesis, Department of Computer Science, Rensselaer Polytechic Institute, February 1992.

[27] M. T. Heath, E. Ng, and B. W. Peyton, *Parallel Algorithms for Sparse Linear Systems*, in Parallel Algorithms for Matrix Computations, Society for Industrial and Applied Mathematics, 1991, pp. 83–124.

[28] B. Hendrickson and R. Leland, *An Improved Spectral Graph Partitioning Algorithm for Mapping Parallel Computations*, Tech. Rep. SAND 92-1460, Sandia National Laboratories, September 1992.

[29] ——, *A Multilevel Algorithm for Partitioning Graphs*, Tech. Rep. SAND 93-1301, Sandia National Laboratories, October 1993.

[30] ——, *Multidimensional Spectral Load Balancing*, Tech. Rep. SAND 93-0071, Sandia National Laboratories, January 1993.

[31] ——, *The Chaco User's Guide Version 1.0*, Tech. Rep. SAND 93-2339, Sandia National Laboratories, October 1993.

[32] M. Hestenes and E. Stiefel, *Methods of Conjugate Gradients for Solving Linear Systems*, Journal of Research National Bureau of Standards, 49 (1952), pp. 409–436.

[33] Y. F. Hu and R. J. Blake, *Numerical Experiences with Partitioning of Unstructured Meshes*, Parallel Computing, 20 (1994), pp. 815–829.

[34] K. Hwang and F. A. Briggs, *Computer Architecture and Parallel Processing*, Mc-Graw Hill Book Company, New York, NY, first ed., 1984.

[35] International Business Machines Corporation, *The RS/6000 SP Specification Sheet*, Tech. Rep. http://www.austin.ibm.com/cgi-bin/systems/sp2.pl (Visited October, 1996), International Business Machines Corporation, July 1996.

[36] R. Jain, *The Art of Computer Systems Performance Analysis: Techniques for Measurement, Simulation, and Modeling*, John Wiley and Sons, New York, first ed., 1991.

[37] D. S. Johnson, C. R. Aragon, L. A. McGeoch, and C. Schevon, *Optimization by Simulated Annealing: An Experimental Evaluation; Part I, Graph Partitioning*, Operations Research, 37 (1989), pp. 865–892.

[38] A. B. Kahng, *Fast Hypergraph Partition*, in Proceedings of the 26th ACM/IEE Design Automation Conference, 1989, pp. 762–766.

[39] G. Karypis and V. Kumar, *METIS: Unstructured Graph Partitioning and Sparse Matrix Ordering System, Version 2.0.* http://www.cs.umn.edu/ karypis/metis/manual.ps (Visited October, 1996), August 1995.

[40] ——, *Multilevel k-way Partitioning Scheme for Irregular Graphs*, Tech. Rep. TR-95-064, University of Minnesota Computer Science Dept., August 1995.

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

[42] B. Lisper and J. Collard, *Extent Analysis of Data Fields*, Tech. Rep. TRITA-IT-9403, Swedish Royal Institute of Technology, January 1994.

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

[44] B. Noble and J. W. Daniel, *Applied Linear Algebra*, Prentice-Hall, Inc., Englewood Cliffs, NJ, second ed., 1977.

[45] Ohio Supercomputing Center, *MPI Primer/Developing with LAM*, Tech. Rep. ftp://ftp.osc.edu/pub/lam/lam60.doc.ps, The Ohio State University, December 1995.

[46] M. Papakhian, *Hardware Configuration of the STARRS/SP Systems.* Document available at http://browndwarf.ucs.indiana.edu/STARRS.hardware.html (Visited October, 1996), October 1996.

[47] R. Parsons and D. Quinlan, *Run-time Recognition of Task Parallelism Within the P++ Parallel Array Class Library*, in Proceedings of the Workshop of Scalable Libraries Conference, Mississippi State University, 1993.

[48] A. Pothen, H. D. Simon, and K.-P. Liou, *Partitioning Sparse Matrices with Eigenvalues of Graphs*, SIAM Journal of Matrix Analysis and Applications, 11 (1990), pp. 430–452.

[49] P. C. Robertson, *Visualizing Color Gamuts: A User Interface for the Effective Use of Perceptual Color Spaces in Data Displays*, Computer Graphics and Applications, 8 (1988), pp. 50–64.

[50] Y. Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, Boston, MA, first ed., 1996.

[51] Y. Saad and M. Schultz, *GMRES: A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems*, SIAM J. Sci. Stat. Comput., 7 (1986), pp. 856–869.

[52] J. R. Shewchuk, *An Introduction to the Conjugate Gradient Method Without the Agonizing Pain*, Tech. Rep. CMU-CS-94-125, Carnegie Mellon University, March 1994.

[53] H. Shiraishi and F. Hirose, *Efficient Placement and Routing for Masterslice LSI*, in Proceedings of the 17th Design Automation Conference, 1980, pp. 458–464.

[54] Silicon Graphics Inc., *The Power Challenge Technical Report*, Tech. Rep. http://www.sgi.com/Products/software/PDF/pwr-chlg (Visited June, 1996), Silicon Graphics, Inc., May 1994.

[55] H. S. Stone, *High Performance Computer Architecture*, Addison-Wesley, Reading, MA, second ed., 1990.

[56] The Message Passing Interface Forum, *MPI: A Message-Passing Interface Standard*, Tech. Rep. UT-CS-94-230, University of Tennessee, Knoxville, May 1994.

[57] ——, *MPI-2: Extensions to the Message-Passing Interface.* http://www.cs.wisc.edu/lederman/mpi2/mpi2-report.ps.Z (Visited October, 1996), October 1996.

[58] H. van der Vorst, *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 (1992), pp. 631–644.

[59] W. Gropp and E. Lusk, *User's Guide for MPICH, a Portable Implementation of MPI.* http://www.mcs.anl.gov/mpi/mpiuserguide/paper.html (Visited October, 1996), February 1996.

[60] R. Williams, *Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations*, Concurrency, 3 (1991), pp. 457–481.

[61] A. Yeremin and L. Kolotilina, *On a Family of Two–Level Preconditionings of the Incomplete Block Factorization Type*, Sov. J. Numer. Anal. Math. Modeling, 1 (1986), pp. 293–320.

[62] Zhichen Xu, Xiaodong Zhang, and Lin Sun, *Semi-Empirical Multiprocessor Performance Predictions*, Tech. Rep. TR-96-05-01, University of Texas, San Antonio, High Performance Comp. and Software Lab., 1996.

[63] H. B. Zhou, *Two Stage m-way Graph Partitioning*, Parallel Computing, 19 (1993), pp. 378–406.

Curriculum Vitae

Thomas Loos was born in Omaha, Nebraska on August 31, 1964. He received his B.S. in Computer Science from the University of Nebraska in 1986 and his M.S. in Computer Studies from North Carolina State University in 1987.