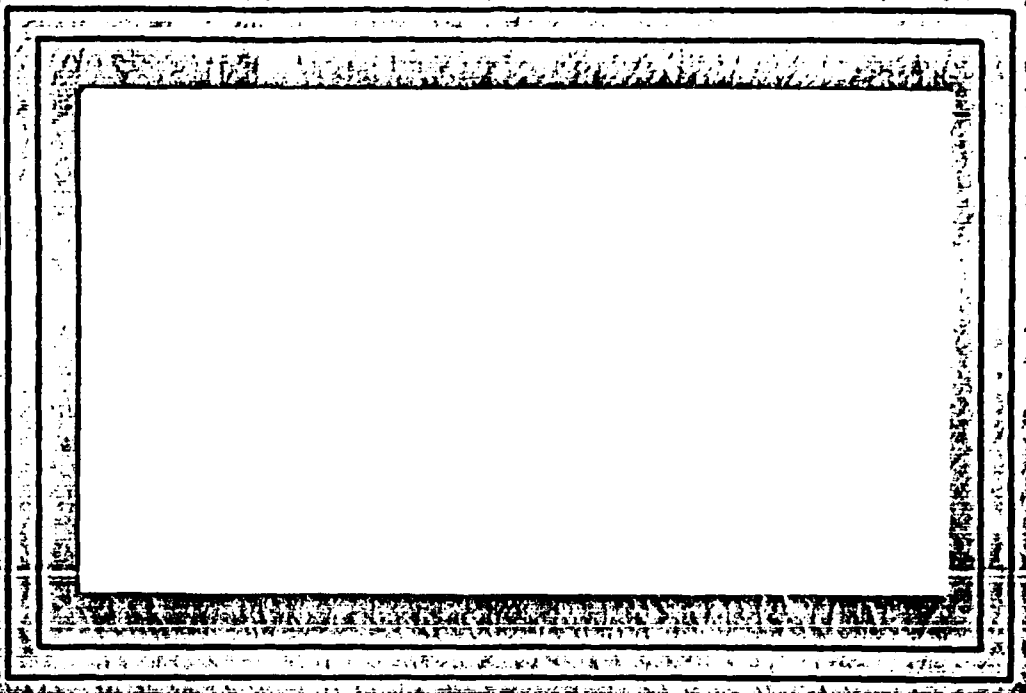

COMPUTER SCIENCE TECHNICAL REPORT SERIES

UNIVERSITY OF MARYLAND
COLLEGE PARK, MARYLAND
20742

UMIACS–TR–88–35 May, 1988
CS–TR–2030

Some Aspects of Parallel Implementation of the Finite Element Method on Message Passing Architectures†

I. Babuška
Department of Mathematics and
Institute for Physical Science and Technology

H. C. Elman
Institute for Advanced Computer Studies and
Department of Computer Science
University of Maryland
College Park, MD 20742

ABSTRACT

We discuss some aspects of implementing the finite element method on parallel computers with local memory and message passing. In particular, we compare the costs of using high order and low order elements, and of direct and iterative solvers for solving the linear systems that occur. Our model of parallel computation is a two–dimensional grid of processors chosen to be similar in shape to the underlying grid. Our main conclusions are that sparse direct solvers generalize naturally to methods based on high order elements, and that direct solvers are adequate for two–dimensional problems, especially for multiple load vectors. We also demonstrate that high order methods achieve higher accuracy at less cost on some typical model problems.

† The work presented in this paper was supported the National Science Foundation under grant DMS–8607478 and by the U.S. Office of Naval Research under contract N–00014–85–K–0169.

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A198736

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER CS-TR-2030 UMIACS-TR-88-35	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE (and Subtitle) Some Aspects of Parallel Implementation of the Finite Element Method on Message Passing Architectures	5. TYPE OF REPORT & PERIOD COVERED Final life of the contract
6. PERFORMING ORG. REPORT NUMBER	
7. AUTHOR(s) [1]I. Babuska and [2]H.C. Elman	8. CONTRACT OR GRANT NUMBER(s) DMS-8607478 NSF N-00014-85-K-0169 ONR
9. PERFORMING ORGANIZATION NAME AND ADDRESS [1]Depart. of Math and Institute for Physical Science and Technology [2]Inst. for Advanced Computer Studies & Dept. of Computer Science University of MD College Park, MD	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Department of the Navy Office of Naval Research Arlington, VA 22217	12. REPORT DATE May 1988 13. NUMBER OF PAGES 28
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release: distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

We discuss some aspects of implementing the finite element method on parallel computers with local memory and message passing. In particular, ~~we compare~~ the costs of using high order and low order elements, and of direct and iterative solvers for solving the linear systems that occur. Our model of parallel computation is a two-dimensional grid of processors chosen to be similar in shape to the underlying grid. Our main conclusions are that sparse direct solvers generalize naturally to methods based on hgih order elements, and that direct solvers are adequate for two-dimensional problems, especially for multiple load

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-6601

(over)

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

vectors. We also demonstrate that high order methods achieve higher accuracy at less cost on some typical model problems.

Abstract. We discuss some aspects of implementing the finite element method on parallel computers with local memory and message passing. In particular, we compare the costs of using high order and low order elements, and of direct and iterative solvers for solving the linear systems that occur. Our model of parallel computation is a two-dimensional grid of processors chosen to be similar in shape to the underlying grid. Our main conclusions are that sparse direct solvers generalize naturally to methods based on high order elements, and that direct solvers are adequate for two-dimensional problems, especially for multiple load vectors. We also demonstrate that high order methods achieve higher accuracy at less cost on some typical model problems.

1. Introduction

The finite element method is a major computational tool of engineering. However, large problems, especially in three dimensions, are not practically solvable on present-day serial computers. Improvements in computing technology, based on the emergence of large-scale parallel architectures, offer the possibility of expanding the set of problems that can be solved effectively. Such machines appear particularly attractive for finite element algorithms, in which the local elements provide a natural decomposition of the problem into (partially) independent sets. The efficiency of any numerical algorithm implemented on a parallel computer depends on both arithmetic and communication, and many aspects of finite element computations, most notably those involving linear algebra, are not fully parallel. In this paper, we address the issues associated with finite element analysis on parallel computers in a simplified way, hoping to get insight into how to use such machines effectively. We focus on two basic issues:

1. a comparison between high order and low order basis functions;
2. a comparison between direct methods and iterative methods for solving the linear algebraic systems that arise.

We consider three versions of the finite element method: the standard h-version, which uses low order basis functions and achieves accuracy by refining the mesh; the p-version, which uses a fixed mesh and achieves accuracy by using high order basis functions; and the hp-version, which combines these two approaches. See [2] for a survey of results for the latter two methods. All these techniques produce a set of one or more linear systems of equations where the coefficient matrix is the global stiffness matrix. Our strategy for solving these systems in parallel is to partition the problem among the available processors and apply local elimination inside each processor, so that unknowns "interior" to processors are decoupled from the system. For computing the other unknowns, we examine both direct solvers based on the parallel nested dissection method [5,6,17] and parallel versions of the conjugate gradient method (CG). The latter strategy is related to domain decomposition methods, for which parallel implementations using "fast direct" local elimination have been considered in [9,10].

For all these techniques, we perform an analysis of the computational complexity of their parallel implementation, and we perform a series of numerical experiments that determine the accuracy of finite element solutions of model problems for various choices of basis functions. Combining the analytic and numerical results gives estimates of the

overall parallel costs. Our model of parallel architecture is a two-dimensional $k \times k$ grid of processors with nearest neighbor connections. Each processor has access to its own local memory, and data can be communicated only between neighboring processors.

An outline of the paper is as follows. In Section 2, we describe the model problem and the finite element methods used for discretization, and in Section 3, we give an overview of the computations needed for solution. In Section 4, we present a cost analysis of the solution techniques, including direct local elimination and global elimination by both direct methods and iterative methods. In Section 5, we combine these results with the results of numerical experiments determining the effectiveness of various finite element methods and iterative solvers to assess the overall costs of solution techniques, and in Section 6 we draw conclusions.

2. The Model Problem and Finite Element Discretizations

Consider the model problem

$$-\sum_{j,k=1}^{2} \frac{\partial}{\partial x_j} a_{jk} \frac{\partial u_i}{\partial x_k} = f_i \text{ on } \Omega, \qquad i = 1, \ldots, s, \tag{2.1a}$$

$$\frac{\partial u_i}{\partial n} = g_i \text{ on } \partial\Omega, \tag{2.1b}$$

where Ω is a square domain, $a_{jk} = a_{kj}$ and the multiload f_i, g_i satisfy the usual solvability conditions. We will cast the problem into the standard variational form: to seek $u_i \in H^1(\Omega)$ so that

$$B(u_i, v) = F_i(v) \tag{2.2}$$

holds for all $v \in H^1(\Omega)$, where

$$B(u, v) = \sum_{j,k=1}^{2} \int_\Omega a_{jk} \frac{\partial u}{\partial x_j} \frac{\partial v}{\partial x_k} \, dx, \tag{2.3a}$$

$$F_i(v) = \int_\Omega f_i v \, dx + \int_{\partial\Omega} g v \, ds, \qquad i = 1, \ldots, s. \tag{2.3b}$$

The solution exists and is unique up to an additive constant. We pose (2.1) on a square domain, but our arguments apply in general to any domain that is topologically a square.

The finite element method consists of the selection of the subspace $S \subset H^1(\Omega)$ and computation of approximate solutions $u_i(S) \in S$ that satisfy

$$B(u_i(S), v) = F_i(v) \quad \forall v \in S. \tag{2.4}$$

The goal is to obtain $u_i(S)$ such that

$$\|u_i(S) - u_i\|_E \leq \tau, \tag{2.5}$$

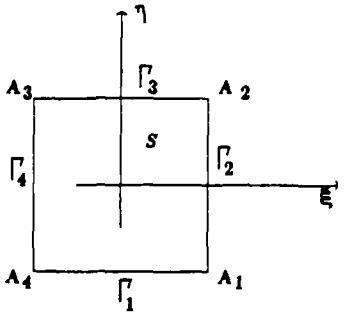

Figure 2.1: The standard square S.

where $\|u\|_E \equiv B(u,u)^{1/2}$ is the energy norm and τ is an a priori given tolerance. One strategy is to use a sequence of spaces $S^{(l)}, l = 1,2,\ldots$ and compute $u_i^{(l)}$ until the result satisfies (2.5). We restrict our attention to the energy norm, although other measures are sometimes more important in practice.

The quality of the finite element solution $u_i(S)$ and the computational work to compute it depend on the regularity of u_i and properties of S, including its dimensionality, which determines the size of the linear system to be solved. For solving (2.1) on a $k \times k$ grid of processors, we will divide Ω into k^2 quadrilateral "super-elements" D_{ij}, $1 \leq i,j \leq k$. Each super-element D_{ij} is the image of the standard square $S \equiv [-1,1] \times [-1,1]$ with vertices A_i and sides Γ_i, $i = 1,\ldots,4$, see Figure 2.1. Let M_{ij} denote the mapping from S onto D_{ij}, which we assume is smooth. We associate with S a finite dimensional space Φ which is the span of basis functions $\{\phi\} = \{\phi^{(0)}\} \cup \{\phi^{(1)}\} \cup \{\phi^{(2)}\}$. Here $\{\phi^{(0)}\}$ are *nodal* shape functions associated with the vertices $\{A_i\}$, $\{\phi^{(1)}\}$ are *side* shape functions associated with the sides $\{\Gamma_i\}$ and $\{\phi^{(2)}\}$ are *internal* shape functions associated with the interior of S. A nodal shape function associated with A_i is zero on the opposite sides Γ_{i-1}, Γ_{i+2}, a side shape function associated with Γ_i is zero on Γ_j, $j \neq i$, and the internal shape functions are zero on $\cup\Gamma_i$. These functions determine a basis for D_{ij} under composition with M_{ij}^{-1}. The basis functions are chosen so that $S \subset H^1(\Omega)$.

Some examples of basis functions $\phi^{(i)}$ are as follows:

(a) *Shape functions for the h-version.* S is divided into triangles as shown in Figure 2.2, left. The shape functions are the piecewise linear "hat functions;" some examples of the supports of such functions are shown in the figure. In an analogous way, we can divide S into squares and use piecewise bilinear shape functions (Figure 2.2, right).

(b) *Shape functions for the p- and hp-versions.* For the p-versions, there are 4 nodal shape functions which are bilinear. For $p \geq 2$ there are $4(p-1)$ side shape functions. For example, for the side $\eta = -1$ the shape function of degree j in ξ is $\phi_j^{(1)}(\xi,\eta) = (\int_{-1}^{\xi} l_j(t)dt)(1-\eta)/2$ where l_j is the Legendre polynomial of order j. The internal shape functions are given by the tensor product of the nodal and side functions. It is possibie to restrict the number of internal shape functions to $\frac{1}{2}(p-2)(p-3)$ for $p \geq 4$ (and zero otherwise), so that the span contains the complete set of polynomials of degree p. This set is used in the commercial code PROBE [16,18]. Hence, there is a total of $4p+$ (for $p \geq 4$) $\frac{1}{2}(p-2)(p-3)$ basis functions. For the hp-version, S is first divided into squares, and the p-version shape

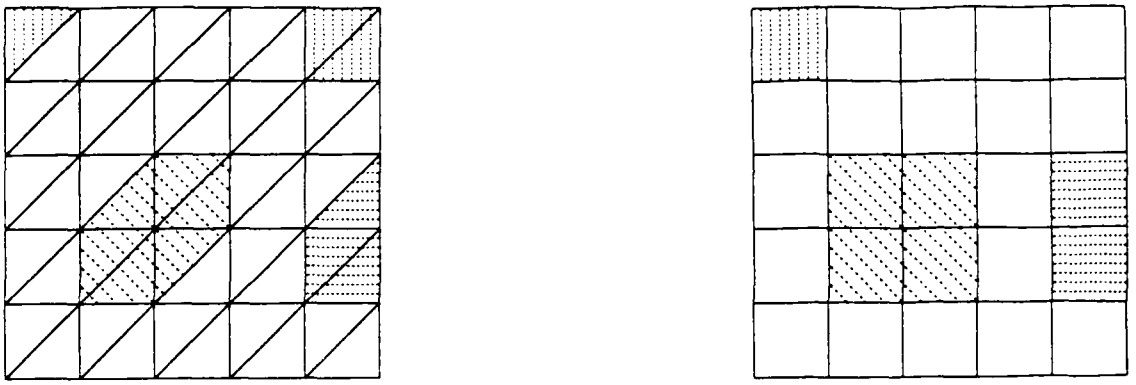

Figure 2.2: Supports of some shape functions for the h-version with triangular and square elements.

functions are used on each square.

(c) *Shape functions for spectral methods* [13,15]. On every side of S, define $p+1$ Lobatto quadrature points ξ_j and polynomials $\theta_i(t)$ such that $\theta_i(\xi_j) = \delta_{ij}$, the Kronecker δ. The shape functions of Φ are then the tensor product functions $\phi_{ij}(\xi,\eta) = \theta_i(\xi)\theta_j(\eta)$. It is straightforward to divide these functions into groups of nodal, side and internal shape functions.

Using the basis functions of Φ, we can easily construct basis functions of S from those defined on $\{D_{ij}\}$. Let these be denoted $\{\psi_j\}$, so that $u_i(S) = \sum_j x_j^{(i)}\psi_j$, $F_i(\psi_j) = y_j^{(i)}$, and (2.4) reduces to a system of linear equations

$$Gx^{(i)} = y^{(i)}, \tag{2.6}$$

where G is the Gramm matrix $[\gamma_{ij}]$ with $\gamma_{ij} = B(\psi_i,\psi_j)$. The unknowns $x^{(i)}$ can be visualized as located in the nodal points, sides and interiors of D_{ij} as in Figure 2.3.

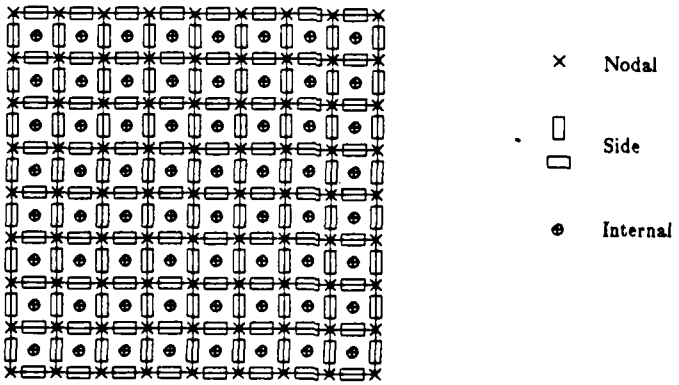

Figure 2.3: Distribution of unknowns on super-elements.

3. Overview of Computations

We would like to compute $u_i(S)$ to the desired accuracy at the lowest possible cost, in terms of work and storage. Thus, we wish to understand the connection between the error

$\|u_i(S) - u_i\|_E$ and the cost of computing $u_i(S)$. This will depend on the choice of S and its basis, the regularity of the solution u_i, the method of linear system solution, and the computer architecture. In this section, we give an overview of the required computations. They can be divided conceptually into the following four steps:

(1) *Computation of local stiffness matrices and local load vector(s).* The Gramm matrix and load vectors of (2.6) are linear combinations of local stiffness matrices (or local load vectors), defined on the individual super-elements D_{ij}. The local stiffness computations entail quadratures to evaluate $B(u,v)$ and $F_i(v)$ on D_{ij}. We will not consider global assembly of G, but instead will examine local eliminations prior to assembly. Hence, these computations are fully parallelizable.

(2) *Local elimination.* The local matrices and load vectors can be written in block form

$$\begin{pmatrix} A & B \\ B^T & C \end{pmatrix}, \qquad \begin{pmatrix} b \\ c \end{pmatrix}. \tag{3.1}$$

A corresponds to the connections among internal unknowns in a super-element, except for those adjacent to the boundary of Ω, where A also includes connections among all side and nodal unknowns not associated with other super-elements. C corresponds to connections among unknowns on the boundary (i.e. nodal and side) D_{ij}, and B corresponds to connections between interior and boundary unknowns. The structure, e.g. sparsity, of A, B and C depends on the choice of shape functions. For example, the h-version produces sparse matrices and the p-version leads to full matrices. For any methods, the interior unknowns can be decoupled from those on the domain interfaces by computing the Schur complement

$$C \leftarrow C - B^T A^{-1} B, \tag{3.2}$$

and modifying all local load vectors similarly by

$$c \leftarrow c - B^T A^{-1} b. \tag{3.3}$$

All these computations are fully parallelizable. For the h-version, explicit local elimination is performed using the serial nested dissection method [7,8], and for the p-version, dense elimination is used.

(3) *Interface solution.* We define the global interface matrix $\hat{G}$ to be the coefficient matrix of the linear system for the unknowns on the domain interfaces after the interior unknowns are decoupled from the system. Interface load vectors g are defined analogously. $\hat{G}$ is computed explicitly by adding components of the local stiffness matrices after local elimination (3.2). The result is one or more systems of linear equations of the form

$$\hat{G}v = g \tag{3.4}$$

which must be solved for the interface unknowns v corresponding to the load vector g.

(4) *Local backsolves.* For any super-element D_{ij}, let v_{ij} denote the component of v on the boundary of D_{ij} determined from the solution of (3.4). The interior values u_{ij} are then obtained by solving (in parallel for all super-elements) the local system $Au_{ij} = b - Bv_{ij}$.

(5) *Postprocessing.* After determining all solutions corresponding to different load vectors, it is usually necessary to determine the values of interest. This is done on an interelement level, and we do not consider it here.

4. Cost Analysis.

In this section, we give a detailed cost analysis for the two-dimensional model problem. For the finite element method, we compare the h-version, the p-version, and the hp-version as discretizations inside super-elements, with m^2 quadrilateral elements for the h- and hp-versions. For solving on the domain interfaces, we compare the costs of direct solution based on a parallel implementation of the nested dissection method, with those of iterative solution using the conjugate gradient method. We will examine CG without preconditioning (although the local elimination could be viewed as a preconditioner), and use of a submatrix of the interface matrix as a preconditioner. We will also distinguish between the cases of one and more than one right hand side.

In our analysis, we state the cost of arithmetic in terms of number of floating point multiplications/divisions. (Additions/subtractions are essentially in one-to-one correspondence with these.) For interprocessor communication, we assume that a processor can communicate with all of its neighbors simultaneously, but at any given moment, data can move in only one direction between two processors. Communication costs are specified in terms of "items of data." We make the simplifying assumption that blocks of data of arbitrary size can be sent (sizes will be chosen to fit algorithms), but each send entails a startup cost. We do not consider overlapping arithmetic and communication.

4.1. Local Computations.

The fully local computations, local stiffness computation (step 1), elimination (step 2), and backsolve (step 4) are performed simultaneously in every processor. For simplicity, we assume that the same discretization is used in every super-element. Let D denote one super-element. For the h- and hp-versions, D is divided into an $m \times m$ grid of local quadrilateral elements, and for the the p-version, no subdivision of D is made.

The local stiffness matrix and load computations are performed by quadratures on D. The complexity depends on the choice of basis functions, but it also depends strongly on the cost of evaluating the coeficients a_{jk} and loads f_i, g_i of (2.1), and, if curved quadrilaterals are used, the Jacobian $J_{M_{ij}}$. Since these costs cannot be stated too precisely, we limit our attention to orders of magnitude.

(a) *The h-version with bilinear elements.* The local stiffness matrix is sparse, symmetric and of order $(m+1)^2$, with 9 diagonals. Each entry comes from $O(1)$ quadratures, so a total of $c_1 m^2$ operations is needed, where c_1 is strongly dependent on the coefficients and Jacobian. Similarly, computation of the load vector requires $c_2 m$ operations where c_2 depends on f and g. We estimate that reasonable values of c_1 and c_2 are $50 - 100$. In special cases (such as constant coefficients), the operation count reduces to essentially zero because the same stencil appears in all elements.

(b) *The p- and hp-versions.* For the p-version ($p \geq 4$), D is not subdivided, and the local stiffness matrix is full of order $Q = 4p+$ (for $p \geq 4$) $\frac{1}{2}(p-2)(p-3)$. Practical

values of p are about $6-10$. To achieve reasonable accuracy when the coefficients a_{ij} or Jacobian is not close to a constant, Gauss quadratures with $(\alpha p)^2$ quadrature points are used where $\alpha > 1$. The computation of the one-dimensional shape functions on αp points needs $c_3 p^2$ operations, and using the tensor product form of the shape functions, the construction of the local stiff matrix needs $c_4 p^4$ operations, where $c_4 \approx \frac{1}{2}$ to 1. In addition, there are $3p^2$ evaluations of coefficients a_{ij} and Jacobians, resulting in a total cost of $c_4 p^2 + c_5 p^2$ operations. It it often the case that $c_5 p^2$ is larger than the first term, so a reasonable estimate for the total cost is cp^4 where $c \approx 1$. For the *hp*-version, these costs are multiplied by m^2, except in the special case of constant coefficients.

The local eliminations are performed by some version of Gaussian elimination, e.g. making use of a factorization $A = LL^T$. It is possible to give precise specification of the costs. For the *p*-method, local elimination can be described as a set of dense block matrix operations, where the blocks are as in (3.1):

Algorithm 1: Local elimination and backsolve.

Elimination:

a) Cholesky factorization $A = LL^T$,

b) Block forward solve $\hat{B} \leftarrow L^{-1}B$,

c) Update $C \leftarrow C - \hat{B}^T\hat{B}$,

d) Forward elimination for each load vector b: $\hat{b} \leftarrow L^{-1}b$ and $c \leftarrow c - \hat{B}^T\hat{b}$,

Backsolve:

e) Given boundary values v, compute interior values $u \leftarrow L^{-T}(\hat{b} - \hat{B}v)$.

Steps (a) - (c) are independent of the number of load vectors; steps (d) and (e) are performed for each load vector. Note that this algorithm must be combined with computation of the interface unknowns, which takes place between steps (d) and (e).

For the *h*-version, the matrices A, B and C are sparse, and we consider use of the nested dissection method, in which the rows and columns of A are symmetrically permuted to minimize fill-in, see [7] for details. (See also Section 4.2 for a parallel version used for elimination on the super-element interfaces.) Formally, the main modification of Algorithm 1 is that step (a) is applied to PAP^T, where P is a permutation matrix that implements the reordering. For the *hp*-method, Algorithm 1 is applied in serial to each element of D to decouple the internal unknowns of local elements from those on local interfaces, and then it is applied using a nested dissection ordering to eliminate the local interfaces.

The following result summarizes the floating point multiplication counts for internal elimination. A proof is given in the Appendix.

Theorem 1: The high order multiplication counts of the local elimination used for two-dimensional problems are

$$m^2\Big(\frac{1}{48}p^6 + \frac{3}{16}p^5 + \frac{17}{16}p^4 - \frac{341}{48}p^3\Big) + \Big(\frac{371}{12}m^3p^3 - \frac{371}{12}m^2p^3 - 17m^2p^2\log_2 m\Big)$$

for factorization, and

$$m^2\Big(\frac{1}{4}p^4 + \frac{3}{2}p^3\Big) + \Big(\frac{31}{2}m^2p^2\log_2 m - 26m^2p + 2m^2\Big) \qquad (4.1)$$

for forward- and back-substitutions.

For both the expressions of this result, the first term in the sum comes from treating the internal unknowns on each of m^2 local elements in D, and the second term comes from treating local element interfaces inside D. Note that the factorization is performed only once, whereas the forward- and back-solves are performed once for each load vector. Also note that for practical values of p, the $O(p^6)$ term is not the governing one.

On a serial computer, all computations are local and costs are summarized the following result.

Theorem 2: The high order multiplication counts for serial factorization are

$$n^2\Big(\frac{1}{48}p^6 + \frac{3}{16}p^5 + \frac{17}{16}p^4 - \frac{341}{48}p^3\Big) + \Big(\frac{267}{28}n^3p^3 - \frac{371}{12}n^2p^3 - 17n^2p^2\log_2 n\Big).$$

The costs are slightly lower than those obtained by replacing m with n in Theorem 1 because the serial algorithm achieves some savings near the boundaries (see [7]). The proof is essentially the same as that of Theorem 1.

4.2. Parallel Direct Solution for Interfaces

After the internal unknowns are decoupled from the system, the result is one or more systems of equations of the form (3.4) whose unknowns are associated only with super-element interfaces. This situation is depicted in Figure 4.1. To simplify the analysis, we assume that the number k of processors in each dimension is a power of two. If D denotes a super-element associated with some processor, let d denote the number of unknowns on each side of D, including one nodal unknown. Thus, each (interior) super-element has $4d$ unknowns associated with it, where unknowns lying on a node are associated with four super-elements, and those on an edge are associated with two. Conceptually, whether these unknowns come from the h-, p- or hp-versions of the finite element method is irrelevant; we simply have $d = mp$ for all methods. The order of $\hat{G}$ is then $N = (2d-1)k^2 + 2kd + 1$. If the unknowns are labeled from 1 to N, then an entry G_{ij} is nonzero iff unknowns i and j are associated with a common subdomain. It has been shown in [5,6,17] that these unknowns can be computed with $O(1)$ efficiency using the parallel nested dissection method. In this section, we give a high-level description of this algorithm and summarize its costs. A detailed description is given in the Appendix.

The method consists of $\log_2 k$ steps, described loosely as follows. At the t'th step, a set of four domains from step $t-1$, each containing $4\delta_t$ boundary unknowns, is merged into a larger domain $D^{(t)}$, and a $\kappa_t \times \kappa_t$ processor grid is used to decouple the interior unknowns of $D^{(t)}$ from the boundary unknowns. (See Figure 4.2 left.) This procedure is repeated recursively, with $\delta_{t+1} = 2\delta_t$ and $\kappa_t = 2^t$. For example, the original k^2 super-element domains $\{D_{ij}^{(0)} \mid 1 \le i,j \le k\}$ (one per processor) each contain $\delta_1 = d$ boundary unknowns on each edge. These k^2 domains are grouped into $(k/2)^2$ square sets containing four domains each, where in each set the four domains are contiguous at one node. Then, simultaneously for each set, the four domains are merged into a larger domain, resulting in a set of $(k/2)^2$ new domains $\{D_{ij}^{(1)} \mid 1 \le i,j \le k/2\}$ each containing $2\delta_1$ boundary unknowns per edge and residing on a $\kappa_1 \times \kappa_1 = 2 \times 2$ grid of processors.

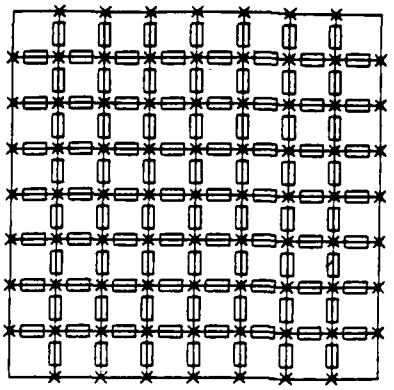

Figure 4.1: Configuration of unknowns after local elimination.

Algebraically, "merging" of subdomains means assembling four stiffness matrices local to the subdomains $\{D^{(t-1)}\}$ into a new stiffness matrix for $D^{(t)}$ (and analogues for load vectors). As in [5], let the unknowns for the merged domain $D^{(t)}$ be divided into ρ_t sets each of size approximately δ_t, where for $1 \leq t \leq \log_2 k - 2$, $\rho_t = 12$. The distribution of unknowns is as in Figure 4.2, left. In the last two steps, there are fewer boundary unknowns (or none at all): $\rho_{\log k - 1} = 8$ and $\rho_{\log k} = 4$. In this discussion, we focus on the case $t \leq \log_2 k - 2$. Let the interior unknowns be labeled with the integers 1 through 4, and the the boundary values labeled 5 through 12. The stiffness matrix for $D^{(t)}$ is then a block 12×12 matrix, denoted by $S^{(t)}$ (Figure 4.2, right). Contributions to each of the blocks of $S^{(t)}$ come from the parts of the local matrices from the previous step whose subdomains are associated with that block. For example, $S_{14}^{(t)}$ contains contributions from the local matrices for subdomain $D_{11}^{(t-1)}$ and subdomain $D_{12}^{(t-1)}$. Merging consists of redistributing the four local stiffness matrices from four separate $\kappa_{t-1} \times \kappa_{t-1}$ grids to one $\kappa_t \times \kappa_t$ grid, and then summing the contributions to $S^{(t)}$.

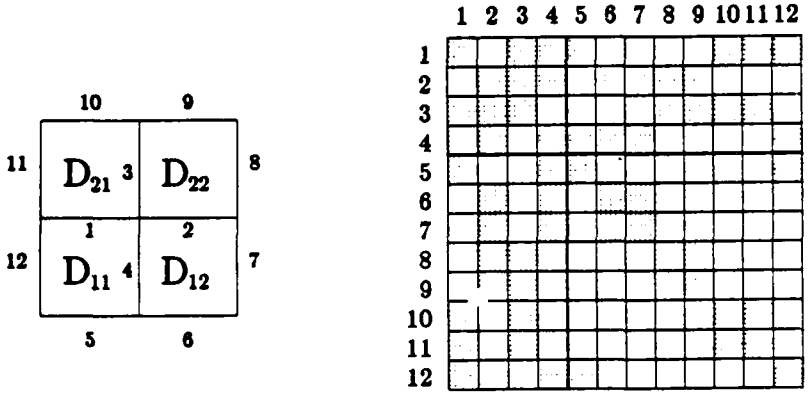

Figure 4.2: A merged domain $D^{(t)}$ and the approximate nonzero structure of its local matrix $S^{(t)}$.

As usual for the nested dissection method, the interior unknowns of each merged domain $D^{(t)}$ form a cross (Figure 4.2, left). To examine the decoupling of the interior points of $D^{(t)}$ from its boundary, we temporarily drop the index t from the discussion, letting D represent a domain at some step $t \leq \log_2 k - 2$. The decoupling is based

on parallel block Gaussian elimination. In an implementation, it would be necessary to identify the twelve sets of unknowns precisely, specifying in particular how points on the boundary of two or more sets (such as the center of the cross) are labeled. We avoid this precise identification, instead deriving an upper bound for the costs by considering a matrix S with a simpler structure: we take S to be a block 12×12 matrix with square blocks whose nonzero blocks are those explicitly identified in Figure 4.2. These blocks are taken to be dense of order δ.* Algorithm 2 is a version of the factorization, block forward solve and update steps (analogues of steps (a)–(c) of Algorithm 1) used to eliminate the first four blocks of S. At each step, it is applied simultaneously to all the local matrices S associated with domains D for that step. The algorithm takes advantage of sparsity and symmetry by operating only on nonzero entries of the block upper triangle of S. At the end of this computation, the lower right 8×8 block is to be merged with three others at the next step.

Algorithm 2: Eliminate a cross for nested dissection.

```
for i = 1 to 4
   factor S_ii into L_ii L_ii^T                          Cholesky factorization
   for j = i + 1 to ρ_t
      if (S_ij ≠ 0) S_ij ← L_ii^{-1} S_ij                Block forward solves
   end
   for j = i + 1 to ρ_j
      for k = j to ρ_t
         if (S_ij ≠ 0 and S_ik ≠ 0) S_jk ← S_jk − S_ij^T S_ik   Matrix-matrix product
      end                                                  (for update)
   end
end
```

The factorization step of the parallel nested dissection is now described by the following algorithm, each step of which is performed on a $\kappa_t \times \kappa_t$ grid of processors:

Algorithm 3: Factorization by parallel nested dissection.

For $t = 1$ to $\log_2 k$

Local assembly: Assemble $S^{(t)}$: merge the four subdomains from step $t-1$ by redistributing and adding four local submatrices.

Eliminate the interior cross: Apply Algorithm 2 with $S = S^{(t)}$.

end

The costs are determined from the individual costs of the merge and the large scale computations of the factorization (Cholesky factorization, block forward solve and, matrix-matrix product) on a $\kappa_t \times \kappa_t$ processor grid. In our complexity analysis (here and in the remainder of the section), we assume that $mp \geq 2$ and $k \geq 4$. The first assumption means that the problem is not very small relative to the number of processors, and the second means that

* The true local matrix for D resembles S, but some diagonal blocks have order $\delta - 1$ instead of δ, and some other off-diagonal blocks have nonzero rows or columns. For example, if the center of the cross is placed in set number 1, then there is a nonzero row in the (1,6) block. It is straightforward to show that the operation counts are higher for this simplified matrix S.

the processor grid contains some interior processors. The following result gives an upper bound for the costs of Algorithm 3; a proof is given in the Appendix.

Theorem 3: The global matrix G can be factored using the parallel nested dissection algorithm with cost

$$\begin{aligned} &\tfrac{73}{4}m^3p^3k - \tfrac{3}{2}m^3p^3\log_2 k - \tfrac{167}{4}m^3p^3 && \text{arithmetic,}\\ &\tfrac{161}{2}m^2p^2k - \tfrac{163}{2}m^2p^2\log_2 k - \tfrac{275}{4}m^2p^2 && \text{communication,}\\ &322k - 326\log_2 k - 275 && \text{startups.} \end{aligned}$$

As in steps (a)–(c) of Algorithm 1, this computation is independent of the number of load vectors. It remains to specify the costs of the global forward elimination and back substitution, which we assume are performed in serial order, once for each load vector. For a single load vector $b = b^{(t)}$ local to $D^{(t)}$, the forward elimination and back substitution are as follows:

Algorithm 4: Forward elimination and back substitution.

Forward elimination.
for $i = 1$ to 4
  $b_i \leftarrow L_{ii}^{-1} b_i$
  for $j = i + 1$ to ρ_t
    if $(S_{ij} \neq 0)$ $b_j \leftarrow b_j - S_{ij}^T b_i$
  end
end

Back substitution
for $i = 4$ to 1
  $b_i \leftarrow 0$
  for $j = i + 1$ to ρ_t
    if $(S_{ij} \neq 0)$ $b_i \leftarrow b_i - S_{ij} b_j$
  end
  $b_i \leftarrow L_i^{-T} b_i$
end

The global eliminations consist of $\log_2 k$ steps of these two computations. We discuss only the forward elimination; the costs for the back substitution are identical.

Algorithm 5: Global forward elimination.

For $t = 1$ to $\log_2 k$
  Local assembly: Assemble $b^{(t)}$: merge the four load vectors from step $t - 1$.
  Eliminate the interior cross: Apply Algorithm 4 with $S = S^{(t)}$ and $b = b^{(t)}$.
end

The following result gives the costs of the forward- and back-solves; see the Appendix for a proof. Note that the asymptotic cost of arithmetic is $O(n^2p^2/k)$, which is suboptimal. (The serial cost is $O(n^2p^2\log_2 n)$, as in the second term of (4.1) with $m = n$.) See e.g. [12] for discussions of efficient parallel triangular solution schemes.

Theorem 4: The cost of forward solves and back substitution for the parallel nested dissection algorithm applied to s load vectors is

$$\begin{aligned} &6m^2p^2k + \tfrac{33s-6}{2}m^2p^2\log_2 k - (18s+6)m^2p^2 && \text{arithmetic,}\\ &33mpk - (38s-30)mp\log_2 k - (28s+26)mp && \text{communication,}\\ &66k + (76s-60)\log_2 k - (56s+52) && \text{startups.} \end{aligned}$$

Finally, the storage requirements for local elimination and parallel nested dissection are outlined in the following result.

Theorem 5: The high order storage requirements per processor for local elimination combined with global elimination by parallel nested dissection are

$$\begin{cases} \left(\frac{1}{8}m^2p^4 + \frac{1}{4}3m^2p^3\right) + \frac{31}{4}m^2p^2\log_2 m & \text{for local elimination,} \\ 30m^2p^2\log_2 k & \text{for global elimination,} \end{cases}$$

where the parenthesized term applies only for $p \geq 4$.

Proof: The local cost comes directly from expression (4.1) of Theorem 1. For the global factorization, it is necessary to store each filled-in version of $S^{(t)}$ computed by Algorithm 2. For all t, the nonzero blocks of $S^{(t)}$ are dense of order δ_t, and they are distributed on a $\kappa_t \times \kappa_t$ grid, so each block requires $\left(\frac{\delta_t}{\kappa_t}\right)^2 = d^2/4$ locations per processor, where $d = mp$. The number of such blocks is 126 for $1 \leq t \leq \log_2 k - 2$, and 54 and 14 for $t = \log_2 k - 1$ and $t = \log_2 k$, respectively. Since both $S^{(t)}$ and $[S^{(t)}]^T$ are used (see the note in the proof of Lemma 1), little advantage is taken of symmetry; the only exception is on the diagonal, where just the lower triangular factor L_{ii} must be stored. Hence,

$$\frac{d^2}{4}\Big(126(\log_2 k - 2) + 54 + 14\Big) - \frac{1}{2}\frac{d^2}{4}\Big(12(\log_2 k - 2) + 8 + 4\Big) = 30d^2\log_2 k - 44.5d^2$$

storage locations are needed in each processor. **Q.E.D.**

Here, we are ignoring pointer overhead and some temporary storage that facilitates pipelining in Algorithm 2 (see the proof of Lemma 2 in the Appendix).

4.3. Parallel Conjugate Gradient for Interfaces

In this section, we outline the costs of a parallel implementation of the conjugate gradient method for solving the global system (3.4). Given an initial guess v_0, CG consists of the following iteration, whose major computations are listed at the right.

Algorithm 6: The preconditioned conjugate gradient method (PCG).

$r_0 \leftarrow g - \hat{G}v_0$, $\tilde{r}_0 \leftarrow M^{-1}r_0$, $p_0 \leftarrow \tilde{r}_0$, $\tau_0 \leftarrow r_0^T\tilde{r}_0$

For $i = 0$ until convergence do

$w_i \leftarrow \hat{G}v_i$	Matrix-vector product
$\eta_i \leftarrow p_i^T w_i$, $\alpha_i \leftarrow \tau_i/\eta_i$	Inner product
$x_{i+1} \leftarrow x_i + \alpha_i p_i$	Scalar-vector product
$r_{i+1} \leftarrow r_i - \alpha_i w_i$	Scalar-vector product
$\tilde{r}_{i+1} \leftarrow M^{-1}r_{i+1}$	Preconditioning
$\tau_{i+1} \leftarrow r_{i+1}^T\tilde{r}_{i+1}$, $\beta_i \leftarrow \tau_{i+1}/\tau_i$	Inner product
$p_{i+1} \leftarrow r_{i+1} + \beta_i p_i$	Scalar-vector product

Thus, each iteration requires one (parallel) matrix-vector product, one preconditioning solve $M\tilde{r} = r$, two inner products and three scalar-vector products. We consider unpreconditioned CG (i.e. where M is the identity operator), as well as using preconditioning by by a sparse approximation of $\hat{G}$.

Our strategy for distribution of data (except the preconditioner) is depicted in Figure 4.3. Each processor P is associated with a super-element D. For any vector v associated

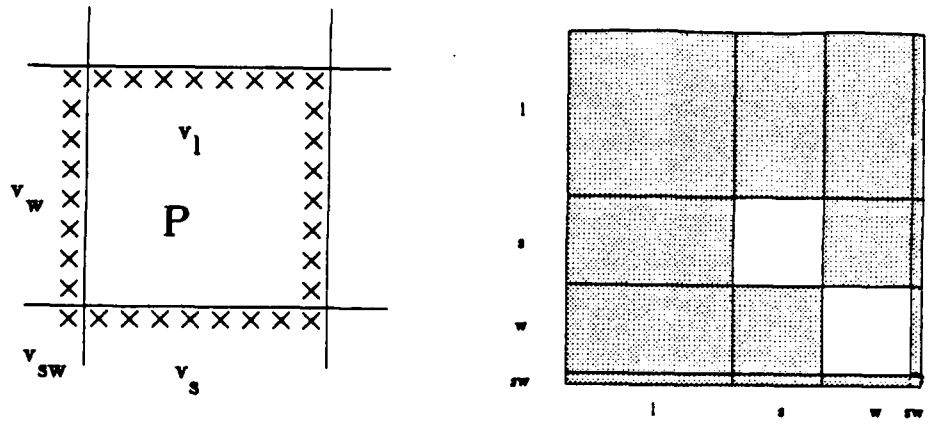

Figure 4.3: Data distribution and local matrix structure for the conjugate gradient method.

with the boundary of D, P contains in its local memory the entries of v on the north and east boundaries, excluding the northwest and southeast entries. For most processors there are $2d-1$ such values, where $d = mp$. (Exceptions are the top row and right column of processors in the grid, which contain no north boundary and no east boundary, respectively. They perform less work than the other processors and are idle some of the time.) The vector v is divided into four disjoint components: v_l for the $2d-1$ entries stored locally in P, and v_w, v_s and v_{sw} for the elements of v stored in the processors to the left, below and below left of P. Let A denote the local matrix for D after local elimination. A is a dense $4d \times 4d$ matrix, but we adopt the convention that the block diagonals of A are assembled to reflect the locations of unknowns. That is, the (s,s), (w,w) and (sw,sw) blocks of A are set to zero, so that there are actually $14d^2 - 1$ nonzeros in A.

The costs of the matrix product and vector operations are as follows (here $d = mp$):
Matrix-vector product $y \leftarrow Ax$: $7d^2$ arithmetic and $4(d+1)$ communication, with 4 startups. This is determined from the following steps of arithmetic and communication, whose costs are listed on the right:

(1) In parallel: send x_w east, x_s north and x_{sw} north.	1 startup, send d words
(2) Send x_{sw} east.	1 startup, send 1 word
(3) Compute $y \leftarrow Ax$	$14d^2 - 1$ arithmetic
(4) Send y_{sw} west.	1 startup, send 1 word
(5) In parallel: send y_w west, y_s south and y_{sw} south	1 startup, send d words

Inner product $\tau \leftarrow \sum_{\text{All domains}} x^T y$: $2d-1$ arithmetic, and $2k$ communication and $2k$ startups to accumulate the sum and distribute it to all processors.
Scalar-vector product $y \leftarrow \alpha x$ for scalar α: $2d-1$ arithmetic and no communication.

For the preconditioner M, we consider a symmetric permutation of the matrix

$$\begin{pmatrix} \tilde{G} & 0 \\ 0 & I \end{pmatrix}, \tag{4.2}$$

where $\tilde{G}$ is a global interface matrix corresponding to a subset of the unknowns on the boundaries of the local interfaces. For example, one could choose the corner unknowns of all super-elements, plus one unknown from each side of every super-element, so that $\tilde{G}$ has the form of a low order operator approximating $\hat{G}$. The permutation maps the

specified unknowns to the lowest indices in the natural way. Suppose $\tilde{d} < d$ unknowns per side are used to define the preconditioner. Then the costs of applying and storing the preconditioning operator are obtained by replacing mp with $\tilde{d}$ in Theorems 4 and 5. Hence, the costs of the conjugate gradient method are as follows.

Theorem 6: The cost per step for CG without preconditioning is

$14m^2p^2 + 10mp - 6$	arithmetic,
$2mp + 4k + 6$	communication,
$4k + 4$	startups.

The additional cost per step for preconditioning by (4.2) is (for $\tilde{d} \geq 2$)

$6\tilde{d}^2k + \frac{27}{2}\tilde{d}^2 \log_2 k - 24$	arithmetic,
$33\tilde{d}k - 8\tilde{d} \log_2 k - 54$	communication,
$66k + 16 \log_2 k$	startups.

The storage requirements (not including those for local elimination) are $14m^2p^2 + 20mp$ without preconditioning (for A, x, r, p and w), and an additional $30m^2 \log_2 k + 4mp$ with preconditioning (for $\tilde{G}$ in factored form and $\tilde{r}$). **Q.E.D.**

There is also a preprocessing cost for factoring $\tilde{G}$, obtained by replacing mp with $\tilde{d}$ in Theorem 3. Note that the efficiency of the unpreconditioned algorithm approaches one as the problem size grows.

An implementation of unpreconditioned CG (with benchmarks on a hypercube) that computes an extra inner product but decreases the startup overhead of inner products is presented in [11].

5. Numerical Experiments

In this section, we describe the results of numerical experiments for solving a model problem, and we combine these results with the cost analysis of the previous section to estimate parallel costs. Consider the model problem

$$-\Delta u = 0 \quad \text{on } \Omega = [0,1] \times [0,1], \tag{5.1a}$$

$$\frac{\partial u}{\partial n}(x_1,0) = \frac{\partial u}{\partial n}(0,x_2) = 0, \qquad \frac{\partial u}{\partial n}(x_1,1) = g_1(x_1), \ \frac{\partial u}{\partial n}(1,x_2) = g_2(x_2), \tag{5.1b}$$

where g_i are determined so that the solution is

$$u(x_1,x_2) = Re\big((a^2+z^2)^{-1} + (a^2-z^2)^{-1}\big) - \frac{2}{a^2}, \qquad a > 1, \quad z = x_1 + ix_2.$$

Note that u is a harmonic function with a singularity at $z = \pm a$, $z = \pm ia$ so that the solution becomes less smooth as $a \to 1$. The parameter a characterizes the regularity of the solution. This model problem is a characteristic one for many problems in structural

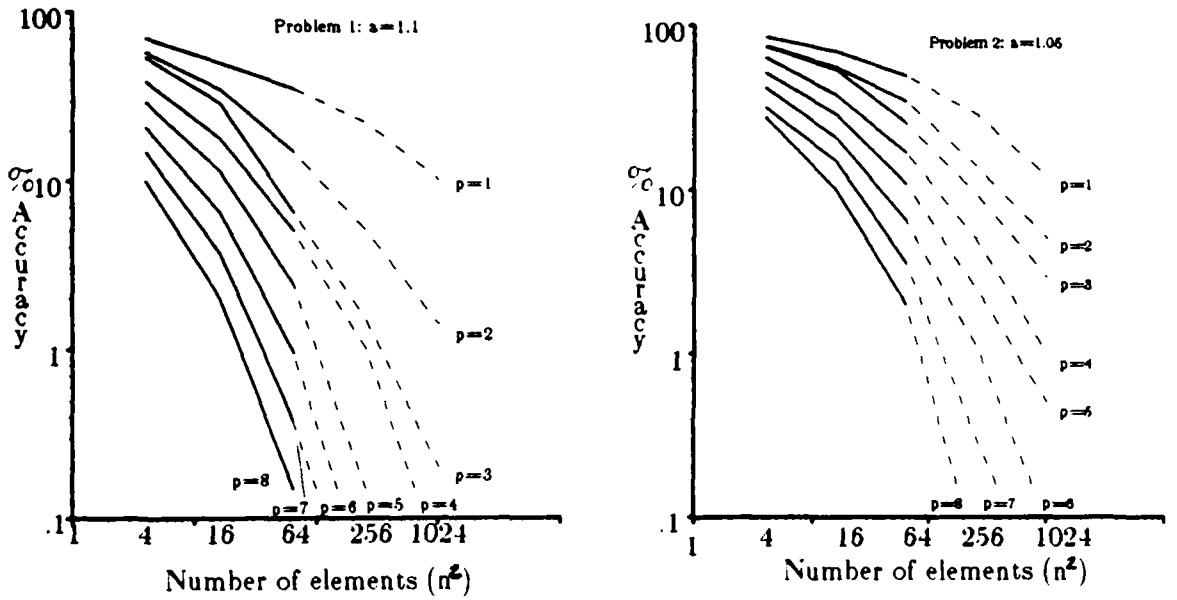

Figure 5.1: Accuracy of the finite element solutions.

mechanics. We consider two versions of this model: Problem 1, with $a = 1.1$, and Problem 2, with $a = 1.05$.

Figure 5.1 shows the accuracy $\|u(S) - u\|_E / \|u\|_E$ obtained using elements of degree p and $n \times n$ element grids with $n = 2$, 4, and 8, to solve (5.1) for the two choices $a = 1.1$ and $a = 1.05$. The finite element computation was made by the code PROBE [16] on an Apollo 3000D in double precision. For use with the parallel analysis of the previous section, we extrapolated these results in two ways to get data for finer grids. First, we used the asymptotic values of the slopes of these curves to extrapolate to values for $n = 16$ and $n = 32$. In the figure, the extrapolated results are indicated by dotted lines. Second, we treated this problem as though it is a subproblem of a larger one discretized on a $4n \times 4n$ element grid, in which Ω is $\frac{1}{16}$ of a larger domain. That is, our operation counts are for a problem posed on a domain with four times as many elements in each direction, but in which the accuracy is as in Figure 5.1. In the following, we consider $n \times n$ element grids with $n = 8$, 16, 32, 64 and 128. Figure 5.2 graphs accuracy as a function of cost for *serial* computations, for direct solves using several choices of finite element method and the two values of a.

For examining parallel costs, unless otherwise specified, we fix the costs of arithmetic and communication as follows. A floating point multiplication is normalized to take one unit of time. Communication is 10 times faster than arithmetic but incurs a startup cost equal to the time required to perform 10 floating point multiplications (or send 100 words). Thus, n words of data can be sent to a neighbor in $10 + .1n$ units of time. (These choices are rough approximations to observed times on both the Intel iPSC and Ncube hypercube parallel processors [12].)

Figures 5.3 – 5.6 show the costs of parallel direct solves for several values of available parameters. Figure 5.3 shows accuracy as a function of cost for direct solution on 64 processors arranged in an 8×8 grid ($k = 8$), for $a = 1.1$ and $a = 1.05$. Comparison

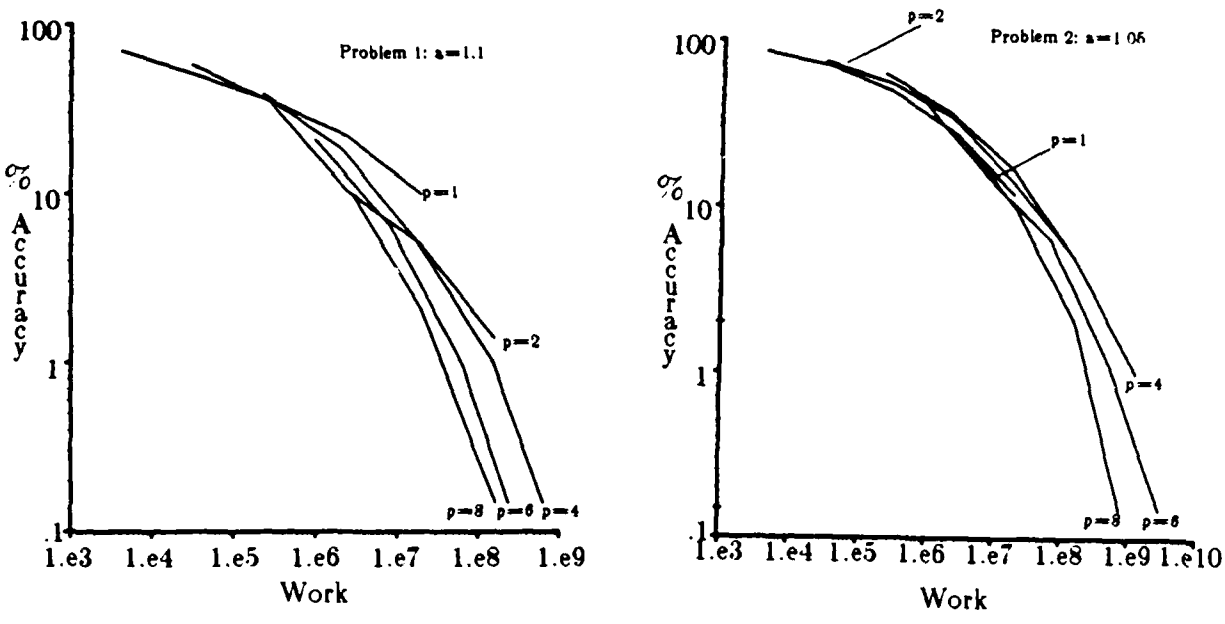

Figure 5.2: Serial costs of finite element solutions.

of Figures 5.2 and 5.3 shows that the relative advantages of choice of basis function are essentially the same in the serial and parallel case, i.e. that high order basis functions achieve greater accuracy. Figure 5.4 shows the costs to achieve accuracy of both 10% and 1%, for values of k ranging from 1 to 64. This figure shows that addition of processors results in decreases in cost until the local problem size becomes too small, after which startup overhead begins to dominate.

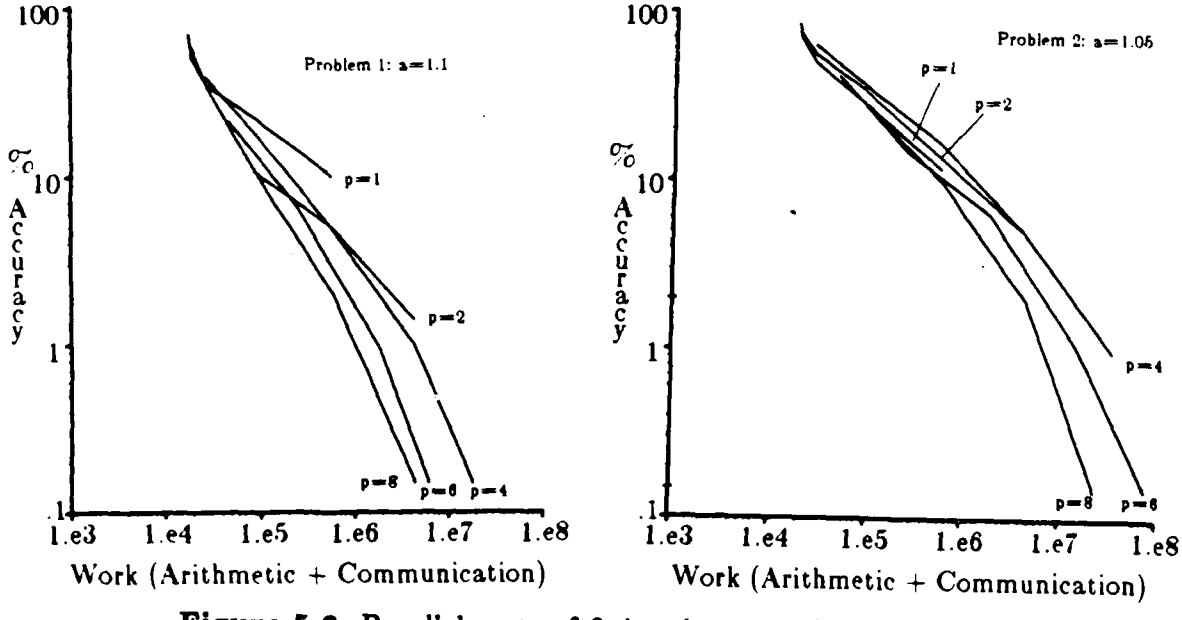

Figure 5.3: Parallel costs of finite element solutions, $k = 8$.

Figure 5.5 shows the speedups achievable for different choices of finite element dis-

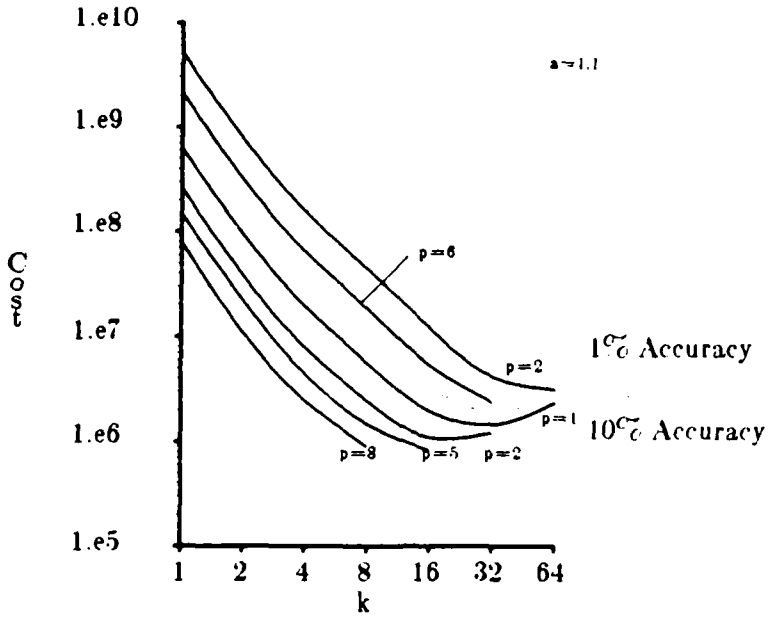

Figure 5.4: Parallel costs for fixed accuracy and varying k (Problem 1).

cretization, for $k = 8$ and $k = 32$. The upper bound on speedup, k^2, is indicated by dashed lines. The maximal speedup is slightly greater than half these upper bounds (giving efficiency of about 50%). There are two reasons that the asymptotic speedup is less than this bound. First, symmetry is not fully exploited in the parallel computations: during the Cholesky factorization of Algorithm 2, computations are duplicated in the upper and lower triangle of the processor grid, and during the block forward solves, only about half the processors are actually computing. (See the Appendix for details.) Second, the parallel dissection cannot take advantage of lower costs near the boundary as well as the serial version; this is reflected in the difference between the coefficient of m^3p^3 in Theorem 3 with that of n^3p^3 in Theorem 2. We also observe that by our convention, more computations are fully local for higher values of p, so that maximal speedup is achieved for smaller element grids for these values.

In Figure 5.6, we vary the relative costs of communication and startups for the same problem considered in Figure 5.3, left (a = 1.1). For Figure 5.6, left, communication speed is decreased to the same speed as arithmetic, with no change in startup cost. For Figure 5.6, right, communication speed is made 10 times faster than arithmetic (as above), but startup cost is decreased to the cost of one multiplication. Comparison with Figure 5.3, left, indicates that the (relatively high) cost of startups significantly degrades performance, even though startups are a lower order overall cost, whereas the cost of communication is less of a factor. This agrees with observations made in [10], although it is also known that for very large problems, communication costs, which are of lower order than arithmetic costs, will be negligable [11].

For the conjugate gradient method, the standard bound on the error at the j'th step has the form [1]

$$\|e^{(j)}\|_E \leq 2(1-\mu)^j \|e^{(0)}\|_E. \tag{5.2}$$

Here, $\|e^{(j)}\|_E = \|u^{(j)} - u(S)\|_E$, the energy norm of the discrete error at the j'th CG

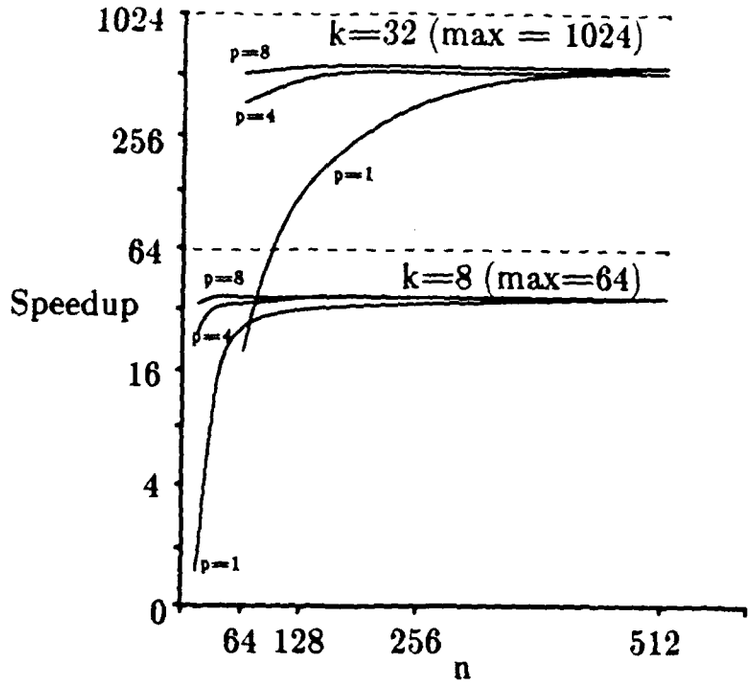

Figure 5.5: Speedups for $k = 8$ and $k = 32$.

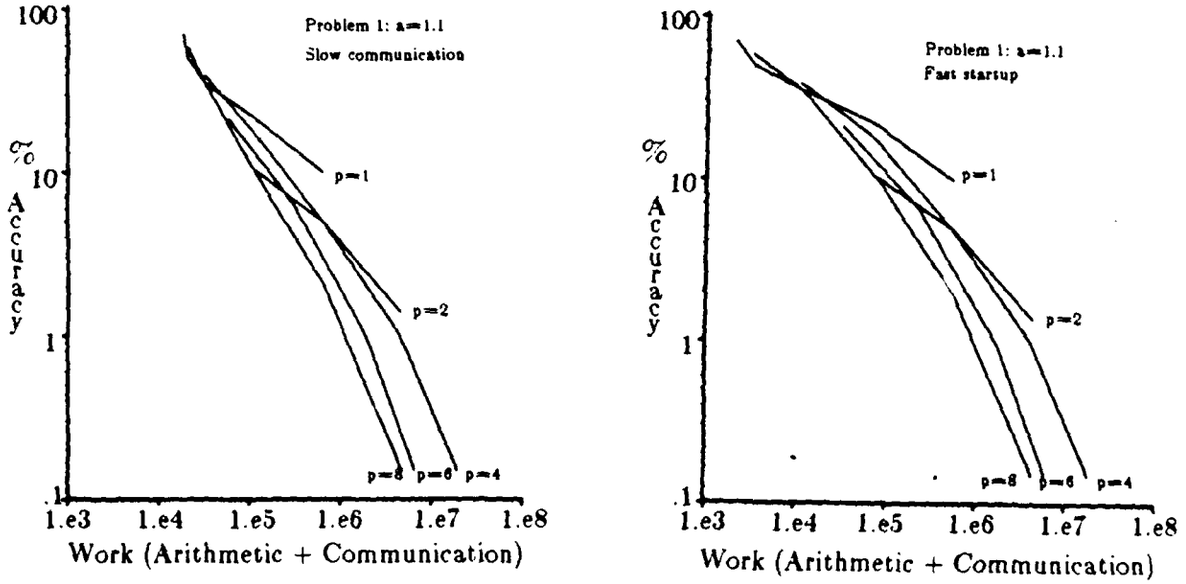

Figure 5.6: Costs for two other models of parallel costs, $k = 8$.

iteration, and the decay factor $1 - \mu$ depends on the iteration matrix. (For uniform eigenvalue distributions, $\mu \approx$ (condition number)$^{-1/2}$.) We study the performance of CG and PCG by examining its performance for $\mu \in (0, 1)$, for solving the two model problems with an accuracy comparable to that achieved with a direct solver.

In particular, let the desired accuracy be 1%. From the data used to produce Figure 5.1, we find that this accuracy is achieved for Problem 1 when (for example) $p = 2$, $n^2 = 1024$, or $p = 6$, $n^2 = 64$, and for Problem 2 when $p = 4$, $n^2 = 1024$ or $p = 8$, $n^2 = 100$. As in our analysis of direct methods, we simulate a finer grid by replacing n

with $4n$. Therefore, we examine the conjugate gradient method for the four choices

Problem 1: $p = 2$, $n = 128$ and $p = 6$, $n = 32$

Problem 2: $p = 4$, $n = 128$ and $p = 8$, $n = 40$.

From (5.2), approximately $j = (\log \epsilon)/\log(1-\mu)$ iterations are needed to bound the relative error $\|e^{(j)}\|_E/\|e^{(0)}\|_E$ by ϵ. We use this estimate on iteration counts, with the choice $\epsilon = 10^{-3}$, to determine the costs of achieving approximately 1% accuracy. Multiplying these iteration counts by the cost per step (taken from Theorem 6) gives the overall cost of the conjugate gradient method. For a preconditioner, we consider the use of a submatrix of the global interface matrix $\hat{G}$ corresponding to the nodal unknowns on the super-elements, plus one side unknown from every side of the super-elements (so that $\tilde{d} = 2$ in Theorem 6).

Figure 5.7 compares the cost of CG and PCG with those of the direct solvers, for solving the two problems with one load vector on an 8×8 processor grid. The results show that if the decay factor $1 - \mu$ is much less than one, then CG and PCG will be more efficient than direct solvers, but that the two classes of methods become comparable in cost as $\mu \to 0$. They also indicate that the overhead for low-order type preconditioners is not a significant extra expense. (For the problems considered, the number of points mp on each interface boundary is at least 40, much larger than $\tilde{d} = 2$.) Figure 5.8 compares the cost of CG and PCG with those of the direct solvers, for solving the two problems with thirty load vectors on an 8×8 grid. Here, the factorization for the preconditioner is counted only once. These results suggest that in the case of multiple load vectors, parallel direct solvers will be highly competitive for two-dimensional problems, even if the rate of convergence of CG or PCG is independent of mesh size. (We remark that synchronization costs for the CG inner products are of low order for the problem sizes considered here.)

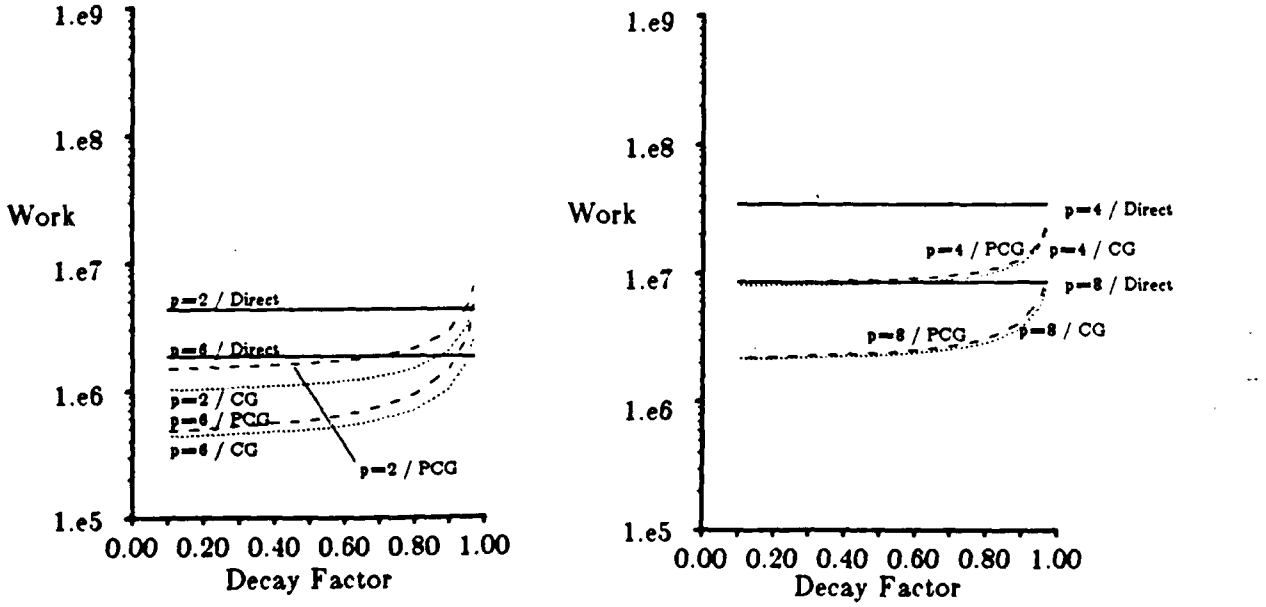

Figure 5.7: Comparison of conjugate gradient and direct solves for one load vector, $k = 8$.

It is known that for $p = 1$, μ behaves like $1/\sqrt{nk}$ [4], so that in this case the unpreconditioned algorithm will have values of $1 - \mu$ near one. We expect $1 - \mu$ to be smaller

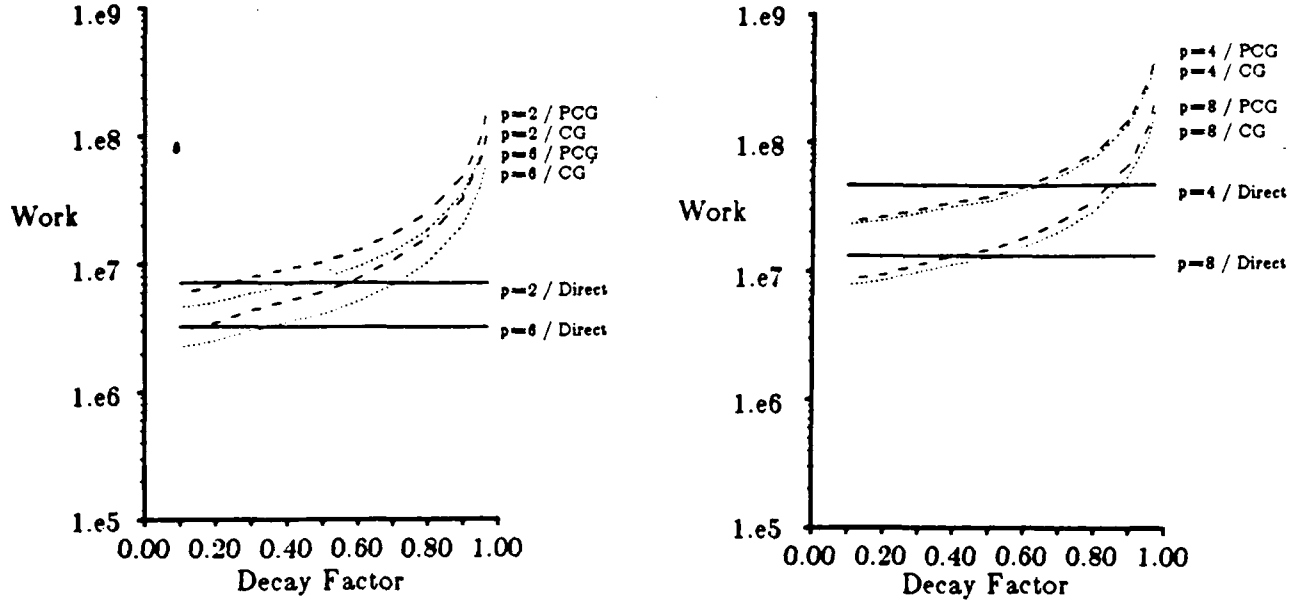

Figure 5.8: Comparison of conjugate gradient and direct solves for thirty load vectors, $k = 8$.

(even without preconditioning) for larger p. To get some insight into this point, we estimated the values of $1 - \mu$ for CG for applied to the discretization of (5.1), both without preconditioning and with a particular low order preconditioner, for several values of n and p. (These estimates are based on the experiments [3].) For each p, the elements and super-elements are the same, and local elimination of the internal unknowns is performed in each element. $\hat{G}$ is the global interface matrix corresponding to the remaining side and nodal unknowns, and the preconditioner $\tilde{G}$ is the submatrix of $\hat{G}$ corresponding to $p = 1$. The results are shown in Table 1. The entries are the average values of $\|e^{(j)}\|_E / \|e^{(j-1)}\|_E$, taken over the first ten iterations of CG, using a zero right hand side and smooth initial guess. The left-hand table is for unpreconditioned CG, and the right-hand table is for preconditioning of the p-version interface operator by the h-version interface operator. Note that the preconditioning is different from that considered above, where local elimination is also applied to unknowns coming from side and nodal unknowns inside super-elements.

	p=2	p=4	p=6	p=8	p=10
n=2	.383	.521	.572	.616	.635
n=4	.595	.644	.672	.699	.714
n=6	.673	.689	.721	.743	.749
n=8	.744	.762	.765	.770	.775
n=10	.794	.808	.805	.804	.806

	p=2	p=4	p=6	p=8	p=10
n=2	0	.473	.537	.574	.614
n=4	.335	.470	.514	.571	.623
n=6	.357	.475	.516	.563	.609
n=8	.354	.472	.512	.554	.600
n=10	.355	.470	.506	.547	.591

Table 1: Average decay factors for CG applied to the p-version interface operator, without preconditioning (left) and with preconditioning by the h-version interface operator (right).

6. Conclusions

We have considered some issues associated with parallel solution of linear problems arising from finite element analysis. Although the quadratures required to set up these systems often occupy a substantial portion of the computational time, these computations are fully parallel; consequently, our focus has been on the costs of linear system solution. The general strategy considered is to divide the problem among available processors, and perform local elimination wherever possible. The unknowns corresponding to "processor interfaces" are then computed using either direct solvers or iterative solvers. Some of the conclusions we've reached are:

1. Sparse direct solvers based on nested dissection are applicable to systems arising from both the h- and hp-version of the finite element method, and the factorization achieves maximum efficiency of approximately 50%.
2. The use of high order elements appears to be a natural way to increase the amount of local computation and achieve accuracy.
3. Sparse direct methods are highly competitive with preconditioned conjugate gradient methods, especially in the case of multiple load vectors. This is despite the fact that triangular system solution does not achieve optimal order speedup.

We have not addressed two critical issues of finite element analysis here, namely three-dimensional problems and adaptivity. Hierarchical direct solvers of the type considered here can be used for three dimensional problems, although the global data movements will be more complex. We speculate that high order methods will be of use in this regime, again by making more of the computations local in a natural way, at the possible cost of increased local storage requirements. A very useful methodology would be effective low order preconditioners for the conjugate gradient method. Adaptive methods are less amenable to the type of analysis considered here, but we believe that methods based on hierarchical use of high order elements would require richer interconnection schemes than the processor mesh considered here.

Appendix

In this section we present proofs for the cost analyses of Sections 4.1 and 4.2. The costs of local elimination are derived from standard analyses of serial direct methods.

Proof of Theorem 1: The operation counts are written in the form $m^2\times$ (cost of p-version) + (cost on local interfaces). The costs for the p-version are derived from the standard analysis of dense elimination [8]: it is known that the number of multiplications required to factor a dense matrix of order α is $\frac{1}{6}\alpha^3 + \frac{1}{2}\alpha^2 - \frac{2}{3}\alpha$. The result follows from the facts that in most elements, A has order $\alpha = \frac{1}{2}(p-2)(p-3)$ and B is a full matrix with $4p$ columns. The result for local interfaces follows directly from George's original analysis of nested dissection [7]. We only consider the case of $k \geq 4$, so that the costs are determined by the (internal) super-elements, which have four boundaries. Hence, in George's terminology, the operation counts are those for elimination of "interior subsets" in the dissection, as in [7], p. 360, Lemma A.2. At the j'th step, for $1 \leq j \leq \log_2 m$, these counts are $(\frac{m}{2^j})^2(\frac{371}{24} \times 2^{3j}p^3 - 17 \times 2^{2j}p^2 - \frac{22}{3} \times 2^j p + 3)$, and the number of nonzeros in

the j'th section of the factors is $(\frac{m}{2^j})^2(\frac{31}{4} \times 2^{2j}p^2 - 13 \times 2^j p + 3)$. The total is obtained by summing these expressions for $j = 1$ to $\log_2 m$. **Q.E.D.**

Now consider the parallel global matrix factorization (Algorithm 3). The analysis presented is a generalization of the analysis of [5]. Data is arranged as follows. Prior to the elimination of the interior cross at the t'th step, $S^{(t)}$ is a block $\rho_t \times \rho_t$ matrix, where for $1 \leq t \leq \log_2 k - 2$, $\rho_t = 12$. $S^{(t)}$ has the the nonzero pattern of Figure 4.2, and only the block upper triangle is needed. Each block of $S^{(t)}$ has order $\delta_t = 2^{t-1}d$ (where $d = mp$), and its δ_t^2 entries are distributed on a $\kappa_t \times \kappa_t$ grid of processors, where $\kappa_t = 2^t$. When it is convenient, we use the symbols S, κ and δ and omit the counter t. The following result summarizes the costs of the large scale computations (Cholesky factorization, block forward solve and matrix-matrix product) that are performed one or more times during each of four steps.

Lemma 1: If the matrix S of Algorithm 2 contains blocks of order δ, and each of the blocks is equidistributed on a $\kappa \times \kappa$ grid of processors, then the individual large scale computations used in a parallel implementation of Algorithm 2 can be implemented with the following costs:

r factorizations:	$(r+2)\kappa - 2$	arithmetic steps
	$(r+2)\kappa - 3$	communication steps
r block forward solves:	$(r+2)\kappa - 2$	arithmetic steps
	$(3r+2)\kappa - (2r+3)$	communication steps
r matrix-matrix products:	$r\kappa$	arithmetic steps
	$2r\kappa - 2r$	communication steps,

where an arithmetic step consists of $\left(\frac{\delta}{\kappa}\right)^3$ multiplications, and a communication step consists of sending $\left(\frac{\delta}{\kappa}\right)^2$ items of data to a neighboring processor.

Proof: We consider the factorizations, block forward solves and matrix products separately. In the proof, individual matrices distributed on the processor grid are indexed according to their locations in the grid, i.e. if A is any such matrix, then $A_{\mu\nu}$ refers to the portion of A stored in the processor indexed by (μ, ν), $1 \leq \mu, \nu \leq \kappa$. The symbols S and L are reserved to refer to the matrices of Algorithm 2, i.e. S_{ij} is a block of S, equidistributed among the processors.

Cholesky factorization. Let $A = MM^T$ denote the Cholesky factorization of a block S_{ii}, $1 \leq i \leq 4$. The factorization moves in a series of waves across the grid where each wave computes one column of M in the lower triangle of the processor grid and, simultaneously, an (identical) row of M^T in the upper triangle (see [14]). The first wave is shown in Figure A.1. Simultaneous steps of arithmetic (a_i) and communication (c_i) are identified by diagonal lines. The steps are synchronized as in the figure, so that the costs of arithmetic are determined by the matrix-matrix products, which require $\left(\frac{\delta}{\kappa}\right)^3$ multiplications; similarly, the communication cost is $\left(\frac{\delta}{\kappa}\right)^2$ words with one startup. The last step of this wave, which takes place in the (κ, κ) (bottom right) processor, is completed after $2\kappa - 1 (= 7)$ steps of arithmetic and $2\kappa - 2 (= 6)$ steps of communication. The second wave is performed on the lower right $(\kappa - 1) \times (\kappa - 1)$ grid of processors, beginning in processor $(2, 2)$ at step a_4. The factorization is completed after $3\kappa - 2$ arithmetic and $3\kappa - 3$ communication steps.

When r factorizations are pipelined (we are concerned only with $r = 1, 2$), it is necessary that the waves of each factorization do not collide with any from an earlier one. This is achieved by having each factorization begin k steps after the previous one, in the $(1,1)$ processor. The cost of r factorizations is then $(r+2)\kappa - 2$ arithmetic steps and $(r+2)\kappa - 2$ communication steps.

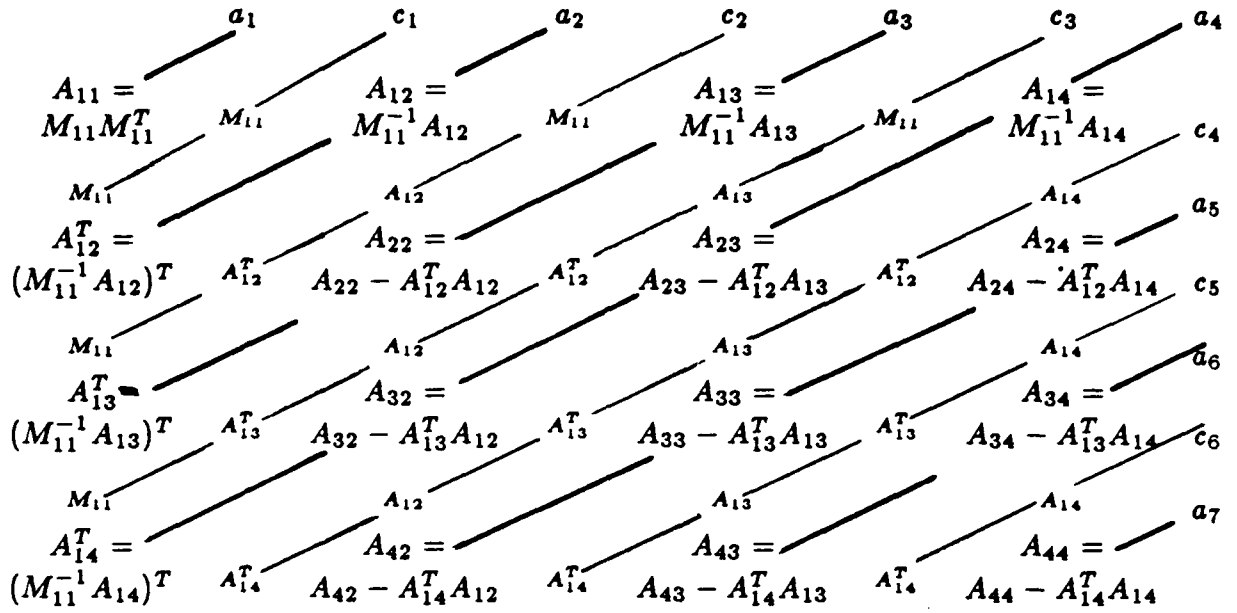

Figure A.1: The first wave of a Cholesky factorizaton on a 4×4 processor grid.

Block forward solve. There are r computations of the form $B = M^{-1}A$, where M is the lower triangular factor distributed in the lower triangle of the processor grid, and A and B are square matrices equidistributed among the processor grid. (In Algorithm 2, $M = L_{ii}$ and $A = S_{ij}$ for some appropriate i, j.) The block solve requires one sweep across the grid for each column of A. Figure A.2 shows the computation of $\{B_{\mu 1} \mid 1 \le \mu \le \kappa\}$, with the result stored in the diagonal processors. $2\kappa - 1$ arithmetic steps and $2\kappa - 2$ communication steps are needed. If all the block columns of A have been positioned in the left processor column, then this step can be pipelined the computation for the i'th column begins at (arithmetic) step i and ends at step $i + 2\kappa - 2$. Similarly, the computation for r block matrices A requires $(r+2)\kappa - 2$ arithmetic steps and $(r+2)\kappa - 3$ communication steps. As above, the cost of each step is $\left(\frac{\delta}{\kappa}\right)^3$ multiplications and $\left(\frac{\delta}{\kappa}\right)^2$ communication with one startup. There is a preprocessing cost of $r(\kappa - 1)$ communication steps to position the columns of A, and a postprocessing cost of $r(\kappa - 1)$ to correctly place the computed columns of B.* The total is $(r+2)\kappa - 2$ arithmetic steps and $(3r+2)\kappa - (2r+3)$ communication steps.

* In the postprocessing step, B_{ij} is moved (horizontally) from the diagonal processor (i,i) to processor (i,j). Since B^T is required for the matrix-matrix product of Algorithm 2, B_{ij} is simultaneously moved (vertically) to processor (j,i).

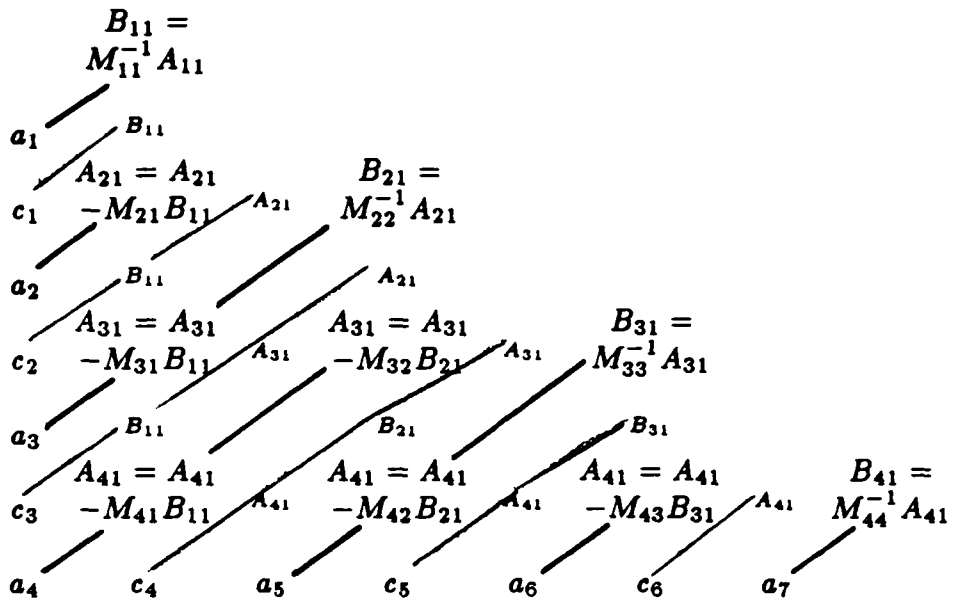

Figure A.2: The block forward solve for the first column on a 4×4 processor grid.

Matrix-matrix product. These computations have the form $B^T A$, which requires a bidirectional horizontal communication of all columns of B^T (see the footnote in the previous paragraph) and, simultaneously, bidirectional vertical communication of all rows of A. This is followed by the block matrix sum $C_{\mu\nu} = \sum_{\sigma=1}^{\kappa} (B_{\sigma\mu})^T A_{\sigma\nu}$ in processor (μ, ν). The cost is $r\kappa$ arithmetic steps and $2r(k-1)$ communication steps. **Q.E.D**

	$1 \le t \le \log_2 k - 2$	$t = \log_2 k - 1$	$t = \log_2 k$
$i = 1$ and 2	2 factorizations 12 solves 42 matrix products	2 factorizations 8 solves 14 matrix products	2 factorizations 4 solves 6 matrix products
$i = 3$	1 factorization 9 solves 45 matrix products	1 factorization 5 solves 15 matrix products	1 factorization 1 solve 1 matrix product
$i = 4$	1 factorization 8 solves 36 matrix products	1 factorization 4 solves 10 matrix products	1 factorization

Table 2: Number of instances of large scale computations in Algorithm 2.

Table 2 shows the number r of times each of the large scale tasks is done. The rows of the table correspond to places where the factorizations and forward solves can be pipelined. Substitution of the values of r from Table 2 into the expressions of Lemma 1 then gives the number of steps required to perform Algorithm 2:

Lemma 2: Under the hypotheses of Lemma 1, the number of steps needed in a parallel implementation of Algorithm 2 is as follows:

Arithmetic steps:	Communication steps:	
$168\kappa_t - 12$	$349\kappa_t - 322$	$1 \le t \le \log_2 k - 2$
$72\kappa_t - 12$	$145\kappa_t - 130$	$t = \log_2 k - 1$
$26\kappa_t - 10$	$43\kappa_t - 39$	$t = \log_2 k.$

Let $T^{(t)}$ denote the lower right square submatrix of $S^{(t)}$ below row 4; for $t \le \log_2 k$, $T^{(t)}$ is an 8×8 block matrix. After the elimination, the block upper triangle of the lower right of $T^{(t)}$ has filled in. Prior to this computation, four such 8×8 block matrices on four separate $\kappa_{t-1} \times \kappa_{t-1}$ processor subgrids are redistributed to the t'th grid and merged into $S = S^{(t)}$. The cost of this operation is summarized by the following result.

Lemma 3: For the matrix S of Lemma 1, the number of communication steps needed to merge the local matrices at step t of Algorithm 3 is

$20\kappa_t$	$t = 1$
$24\kappa_t - 4$	$2 \le t \le \log_2 k - 1$
$8\kappa_t - 4$	$t = \log_2 k.$

Proof: For $t \le \log_2 k - 1$, let $T = T^{(t)}$ denote the 8×8 matrix from step $t-1$ as above, and let the four subdomain quadrants be denoted $\{D_{i,j} \mid 1 \le i,j \le 2\}$ (see Figure 4.2). T is associated with one of the quadrants, wlog, $D_{1,1}$. Each block of T has order $\frac{\delta_t}{2}$ and is distributed on the $\frac{\kappa_t}{2} \times \frac{\kappa_t}{2}$ grid. Now let T be relabeled as a block 4×4 matrix in which each new block is square (of order δ_t) and contains four old blocks. Let U denote one of these new blocks of T; by definition, U can also be thought of as divided into four quadrants. The data movement needed for merging is to move each quadrant of U to the corresponding quadrant of processors for the subdomains. The relocation of U can proceed in the following four steps:

1. Move $U_{1,2}$ from quadrant (1,1) to quadrant (1,2) (clockwise).
2. Move $U_{2,1}$ from quadrant (1,1) to quadrant (2,1) (counterclockwise).
3. Move $U_{2,2}$ from quadrant (1,1) to quadrant (1,2) (clockwise).
4. Move $U_{2,2}$ from quadrant (1,2) to quadrant (2,2) (clockwise).

There are 10 nonzero blocks of the form of U in T, so that 40 movements of this type are needed to relocate all of T. At each step, data must be moved across $\frac{\kappa_t}{2}$ processors, so that the number of communication steps is $20\kappa_t$. Data from the other three quadrants is simultaneously relocated in an analogous manner. This is all the communication required for $t \le \log_k -1$, except that prior to these steps, the lower left quadrant of the (new) diagonal blocks of T (in the $(2,1)$ position) are not explicitly represented when $t > 1$; these quadrants can be made available at cost $4(\kappa_t - 1)$. The same analysis applies for $t = \log_2 k$, except in this case T is a 4×4 block matrix, or 2×2 after relabeling, and only two transpose blocks must be computed. **Q.E.D.**

Proof of Theorem 3: The result is obtained by summing the expressions of Lemmas 2 and 3 for $t = 1$ to $\log_2 k$, and using the values $\delta_t = 2^t d/2$, $\kappa_t = 2^t$, and the facts that each arithmetic step requires $\left(\frac{\delta_t}{\kappa_t}\right)^3 = d^3/8$ multiplications and each communication step requires one startup plus sending of $\left(\frac{\delta_t}{\kappa_t}\right)^2 = d^2/4$ words. **Q.E.D.**

	$1 \le t \le \log_2 k - 2$	$t = \log_2 k - 1$	$t = \log_2 k$
$i = 1$ and 2	$2s$ solves $12s$ matrix products	$2s$ solves $8s$ matrix products	$2s$ solves $4s$ matrix products
$i = 3$	s solves $9s$ matrix products	s solves $5s$ matrix products	s solves s matrix product
$i = 4$	s solves $8s$ matrix products	s solves $4s$ matrix products	s solves

Table 3: Number of instances of large scale computations in Algorithm 4.

The analysis of the global forward and back solution steps is similar. We consider only the forward solve, for which the large scale computations are lower triangular matrix solves and matrix-vector products. Table 3 shows the number of such computations required for s load vectors. Every right hand side $b = b^{(t)}$ at step t is (for $t \le \log_2 k - 2$) a vector with 12 blocks of size δ_t; assume that each of these blocks is distributed among the diagonal processors of the $\kappa_t \times \kappa_t$ grid. If there are s such right hand sides, then the cost of the forward elimination is determined from the following result.

Lemma 4: On a $\kappa \times \kappa$ processor grid, the individual large scale computations of Algorithm 4 can be implemented with the following costs:

rs forward solves:	$2\kappa + rs - 2$	arithmetic steps
	$4\kappa + 3rs - 7$	communication steps
rs matrix-vector products:	rs	arithmetic steps
	$2\kappa + s - 2$	communication steps $(i = 1, 2, 3)$
	$2\kappa + rs - 2$	communication steps $(i = 4)$.

where an arithmetic step consists of $\left(\frac{\delta}{\kappa}\right)^2$ multiplications and a communication step consists of sending $\frac{\delta}{\kappa}$ items of data to a neighboring processor.

Proof: The operations and costs required for rs forward solves are as follows.

(1) Relocate rs vectors from the diagonal processors to the leftmost processors in the grid. With pipelining, the cost is $\kappa + rs - 2$ communication steps.

(2) Compute rs solves of the form $b \leftarrow M^{-1}b$. Here, M is some triangular factor L_{ii} produced by Algorithm 3 (see the proof of Lemma 1). The cost is $2\kappa + rs - 2$ arithmetic steps and $2\kappa + 4s - 3$ communication steps. Each resulting b is located in the diagonal processors.

(3) Distribute each b across processor columns, i.e. move a copy of the part of b in the (μ, μ) processor to each processor (μ, ν), $1 \le \nu \le \kappa$. The cost is $\kappa + rs - 2$ communication steps.

The rs matrix-vector products then require rs arithmetic steps, using the matrices S_{ij}^T constructed by Algorithm 3. The results must ultimately be summed across the processor rows and stored in the diagonal processors. At step i of Algorithm 4, $1 \le i \le 3$, it is only necessary to do this for each (of s versions of) b_{i+1}, which requires $\kappa + s - 2$ communication steps. At step $i = 4$, rs accumulations are required, at cost $\kappa + rs - 2$ communication steps. **Q.E.D.**

Lemma 5: The number of steps needed in a parallel implementation of Algorithm 5 is as follows:

Arithmetic steps:	Communication steps:	
$6\kappa_t + 33s - 6$	$15\kappa_t + 22s - 27$	$1 \le t \le \log_2 k - 2$
$6\kappa_t + 21s - 6$	$15\kappa_t + 18s - 27$	$t = \log_2 k - 1$
$6\kappa_t + 9s - 6$	$14\kappa_t + 14s - 23$	$t = \log_2 k.$

Proof: Substitute the values of r from Table 3 into the expressions of Lemma 4. **Q.E.D.**

Lemma 6: The number of communication steps needed to merge s local load vectors at step t of Algorithm 5 is

$$\begin{array}{ll} 3\kappa_t/2 + 16s - 3 & 1 \le t \le \log_2 k - 1 \\ 3\kappa_t/2 + 8s - 3 & t = \log_2 k. \end{array}$$

Proof: We only discuss the case $1 \le t \le \log_2 k - 1$. Before the elimination step there are four quadrants of $\frac{\kappa_t}{2} \times \frac{\kappa_t}{2}$ processor grids. For every load vector b, there are eight blocks $\{b_j\}_{j=5}^{12}$ distributed among the diagonal processors in each of these quadrants. The stategy for merging is to move the even numbered blocks b_6, b_8, b_{10}, b_{12} down, to the diagonal processors of the lower (southern) quadrants, and the odd numbered blocks b_5, b_7, b_9, b_{11} up (north). The cost is $2(\frac{\kappa_t}{2} + 4s - 1)$ communication steps for the bidirectional move. Then, in the north quadrants, all eastern data is moved west to the diagonal blocks, and in the south quadrants, all western data is moved east. The cost is $\frac{\kappa_t}{2} + 8s - 1$ communication steps. **Q.E.D.**

Proof of Theorem 4: Sum the expressions of Lemmas 5 and 6 for $t = 1$ to $\log_2 k$, where each arithmetic step requires $\left(\frac{\delta_t}{\kappa_t}\right)^2 = d^2/4$ multiplications and each communication step requires one startup plus sending of $\frac{\delta_t}{\kappa_t} = d/2$ words. **Q.E.D.**

References

[1] O. Axelsson, Solution of linear systems of equations: iterative methods, in V. A. Barker, Ed., *Sparse Matrix Techniques*, Springer-Verlag, New York, 1976, pp. 1-51.

[2] I. Babuška, The p and $h - p$ Versions of the Finite Element Method: The State of the Art. Tech. Note BN-1056, Institute for Physical Science and Technology, University of Maryland, 1986.

[3] I. Babuška and J. Pitkaranta, personal communication, 1988.

[4] P. E. Bjørstad and O. B. Widlund, Iterative methods for the solution of elliptic problems partitioned into substructures, *SIAM J. Numer. Anal.* 23: 1097-1120, 1986.

[5] J. M. Conroy, Parallel Direct Solution of Sparse Linear Systems of Equations, Ph.D. Thesis, University of Maryland, 1986. Available as Computer Science Technical Report 1714.

[6] D. Gannon, A note on pipelining a mesh connected multiprocessor for finite element problems by nested dissection, in *Proceedings of the 1980 Intl. Conf on Parallel Processing,*

IEEE Catalog No. 80CH1569-3, pp. 197-204, 1980.

[7] A. George, Nested dissection of a regular finite element mesh, *SIAM J. Numer. Anal.* 10:345-363, 1973.

[8] A. George and J. W. Liu, *Computer Solution of Large Sparse Positive Definite Systems*, Prentice-Hall, New Jersey, 1981.

[9] W. D. Gropp and D. E. Keyes, A comparison of domain decomposition techniques for elliptic partial differential elliptic partial differential equations and their parallel implementation *SIAM J. Sci. Stat. Comput.* 8:s166-s202, 1987.

[10] W. D. Gropp and D. E. Keyes, Complexity of parallel implementation of domain decomposition techniques for elliptic partial differential equations, *SIAM J. Sci. Stat. Comput.* 9:312-326, 1988.

[11] J. L. Gustafson, G. R. Montry and R. E. Benner, Development of Parallel Methods for a 1024-Processor Hypercube, SIAM Preprint, to appear in *SIAM J. Sci. Stat. Comput.*, 1988.

[12] M. T. Heath and C. H. Romine, Parallel solution of triangular systems on distributed-memory multiprocessors, *SIAM J. Sci. Stat. Comput.* 9:558-588, 1988.

[13] NEKTON, Nektonic, Inc., Bedford, MA, U.S.A.

[14] D. P. O'Leary and G. W. Stewart, Data-flow algorithms for parallel matrix computations, *Comm. ACM.* 28:840-853, 1985.

[15] A. T. Patera, Advances and future directions of research on spectral methods, *Computational Mechanics: Advances and Trends,* A. K. Noor, Ed., AMD–Vol 75, ASME, 1987, pp. 411-427.

[16] PROBE, Noetic Technologies Inc., St. Louis, MO, U.S.A.

[17] P. H. Worley and R. Schreiber, Nested Dissection on a Mesh-Connected Processor Array, in A. Wouk, ed., *New Computing Environments: Parallel, Vector and Systolic*, SIAM Publications, Philadelphia, 1986, pp. 8-38.

[18] B. A. Szabo, PROBE Theoretical Manual, 1981, Noetic Technologies Inc., St. Louis, MO, U.S.A.