We consider the application of domain decomposition techniques to the solution of sparse linear systems arising from implicit PDE discretizations on parallel computers. Representatives of two popular MIMD architectures, message passing (the Intel iPSC/2-SX) and shared memory (the Encore Multimax 320), are employed. We run the same numerical experiments on each, namely stripwise and boxwise decompositions of the unit square, using up to 64 subdomains and containing up to 64K degrees of freedom. We produce a tight-fitting complexity model for the former and discuss the difficulty of doing so for the latter. We also evaluate which of three types of domain decomposition preconditioners that have appeared in the literature of self-adjoint elliptic problems are most efficient in different regions of machine-problem parameter space. Some form of global sharing of information in the preconditioner is required for efficient overall parallel implementation in the region of most practical interest (large problem sizes and large numbers of processors); otherwise, an increasing iteration count inveighs against the gains of concurrency. Our results on a *per iteration* basis also hold for sparse discrete systems arising from other types of partial differential equations, but in the absence of a theory for the dependence of the convergence rate upon the granularity of the decomposition, the overall results are only suggestive for more general systems.

Domain Decomposition on Parallel Computers

William D. Gropp† and David E. Keyes‡

Research Report YALEU/DCS/RR-723
August 1989

† Department of Computer Science, Yale University, New Haven, CT 06520. The work of this author was supported in part by the Office of Naval Research under contract N00014-86-K-0310 and the National Science Foundation under contract number DCR 8521451.

‡ Department of Mechanical Engineering, Yale University, New Haven, CT 06520. The work of this author was supported in part by the National Science Foundation under contract number EET-8707109.

1. Introduction

Domain decomposition techniques appear to be a natural way to distribute the solution of large sparse linear systems across many parallel processors. In this paper we develop complexity estimates for two types of decompositions and two parameterized types of "real" parallel computer architectures, and validate those estimates on representative machines, with particular emphasis on the case of large numbers of processors and large problems. We examine the tradeoffs between various forms of preconditioning, as characterized by the efficiency of their parallel implementation.

Parallel computers may be divided into two broad classes: distributed memory and shared memory. In a distributed memory parallel processor, each processor has its own memory and no direct access to memory on any other processor. Such machines are usually termed "message passing" computers since interprocessor communication is accomplished through the sending and receiving of messages. In a shared memory parallel processor, each processor has direct, random access to the same memory space as every other processor. Interprocessor communication is conducted directly in the shared memory. In practice, of course, most shared memory machines have local memory, called the cache, and communication is through messages, called cache faults. However, each type of parallel processor is optimized for a different interprocessor communication pattern, and we consider the effects of these optimizations on domain decomposition.

Domain decomposition refers in a generic way to the replacement of a partial differential equation problem defined over a global domain with a series of problems over subdomains which collectively cover the original. Early domain decomposition techniques, whether iterative [12] or direct [10], were based on exact reductions of the global problem to a set of lower-dimensional problems on interfaces between subdomains by means of direct elimination of the degrees of freedom interior to the subdomains. In this sense, domain decomposition is analogous to the finite element procedure of static condensation. A more modern viewpoint has emerged [2, 13] in which the interior unknowns are not directly eliminated, but retained in the outer iteration, preconditioned by an appropriate fast approximate solver. In [8], we referred to such approaches as "partitioned matrix methods", and it is such that we consider here, in combination with preconditioned conjugate gradients as the outer iterative method. Partitioned matrix iteration was shown in [8] to be identical to iteration on the reduced system when the subdomain solves are exact. However, it has the advantage of being much more flexible for problems for which the only known exact solvers are expensive. The important question as to how much approximate subdomain solves "penalize" overall iterative convergence has begun to be addressed (see, in addition to the references above, [1] and section 6 of [4]).

An example of the simplest form of domain decomposition is shown in Figure 1a. A single interface (3) divides a domain into subdomains 1 and 2. The matrix obtained upon finite difference or finite element discretization is partitioned according to the physical decomposition into subdomains connected by a small (lower dimension) interface region and looks like

$$A_s = \begin{pmatrix} A_{11} & 0 & A_{13} \\ 0 & A_{22} & A_{23} \\ A_{13}^T & A_{23}^T & A_{33} \end{pmatrix},$$

where A_{11} and A_{22} come from the interior of the subdomains, A_{33} from *along* the interface, and A_{13} and A_{23} from the interactions between the subdomains and the interfaces. A detailed view of this matrix for the 5-point operator is shown in Figure 2.

The generalization of Figure 1a to any number of strips is straightforward. For reasons that will become clear, it is important to also consider a decomposition into boxes as shown in Figure 1b.

The matrix for this decomposition looks like

$$A_c = \begin{pmatrix} A_I & A_{IB} & 0 \\ A_{IB}^T & A_B & A_{BC} \\ 0 & A_{BC}^T & A_C \end{pmatrix},$$

where

$$\begin{aligned} A_I &= \text{Block diagonal subdomain matrices} \\ A_{IB} &= \text{Coupling to interiors from subdomain interfaces} \\ A_B &= \text{Block diagonal subdomain interfaces} \\ A_{BC} &= \text{Coupling to interfaces from cross points} \\ A_C &= \text{Cross point system} \end{aligned}$$

This representation is based on a five-point difference template, so that the coupling matrix A_{IC} is zero. More generally, nonzero coupling between the crosspoints and the corner points of the subdomain interiors would need to be taken into account. A detailed view of this matrix for the 5-point operator is shown in Figure 3, for the case of a decomposition into two by two blocks. We are interested in various preconditioners for both A_s and A_c, based on their efficacy and on their parallel limitations.

The choice of preconditioner is critical in domain decomposition, as with any iterative method. In the context of parallel computing, the main distinction is between preconditioners which are purely local, those which involve neighbor communication, and those which involve global communication. (We use the word "communication" here in a general sense; in a shared memory machine, this refers to shared access to memory.) As example preconditioners we consider a block diagonal matrix for the purely local case and a preconditioner based on FFT solves along the interfaces for the neighbor communication case. As an example involving global communication, we consider the Bramble *et al* preconditioner [2], which requires the solution of a linear system for the cross points (or vertices). This method involves only low bandwidth global communication (that is, the size of the messages scales with the number of processors); we do not consider any method which uses high bandwidth communication (where message size scales with the size of the problem).

We develop a complexity model for each type of parallel computer that is based on two major contributions: floating point work and "shared memory access". This latter term measures the cost of communicating information between processors. In a distributed memory system, this is represented by a communication time. In a shared memory system, there are several contributions, including cache size and bandwidth, and the number of simultaneous memory requests which may

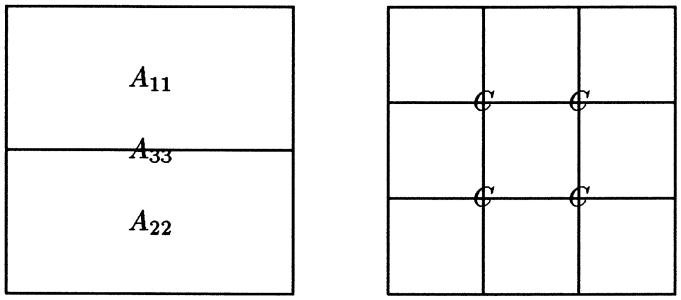

Figure 1: Two forms of decompositions. A two domain strip decomposition is shown in (a); a many domain box decomposition with cross points (labeled "C") is shown in (b).

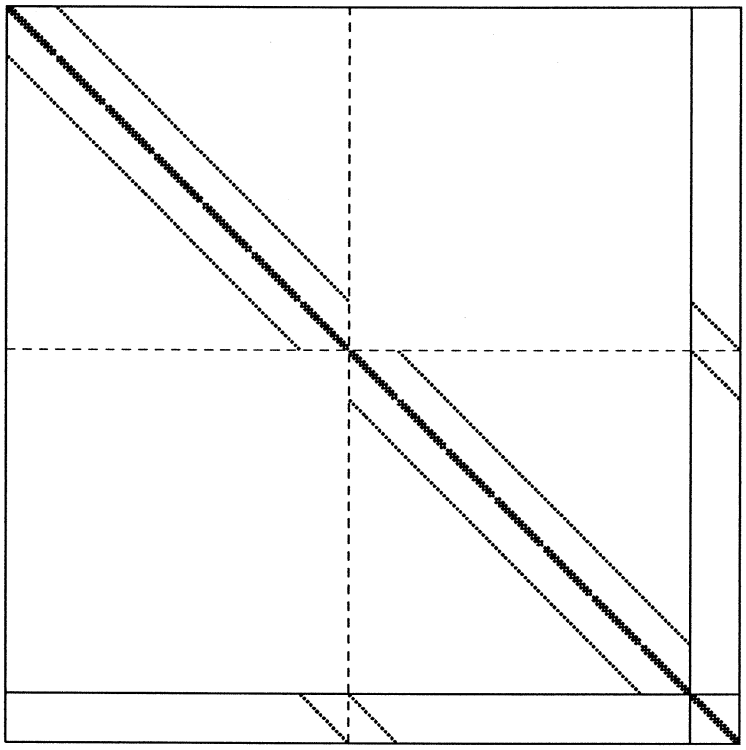

Figure 2: The partitioning of the matrix A_s for the 5-point Laplacian and two strips. The dashed lines separate subdomains; the solid lines separate the interior unknowns from the interface unknowns.

be served.

2. Comments on Parallelism Costs

In any parallel algorithm, there are a number of different costs to consider. The most obvious of these are intrinsically serial computations. For example, the dot products in the conjugate gradient method involve the reduction of values to a single sum; this takes at least $\log p$ time. More subtle are costs from the implementation, both software and hardware. An example of the software

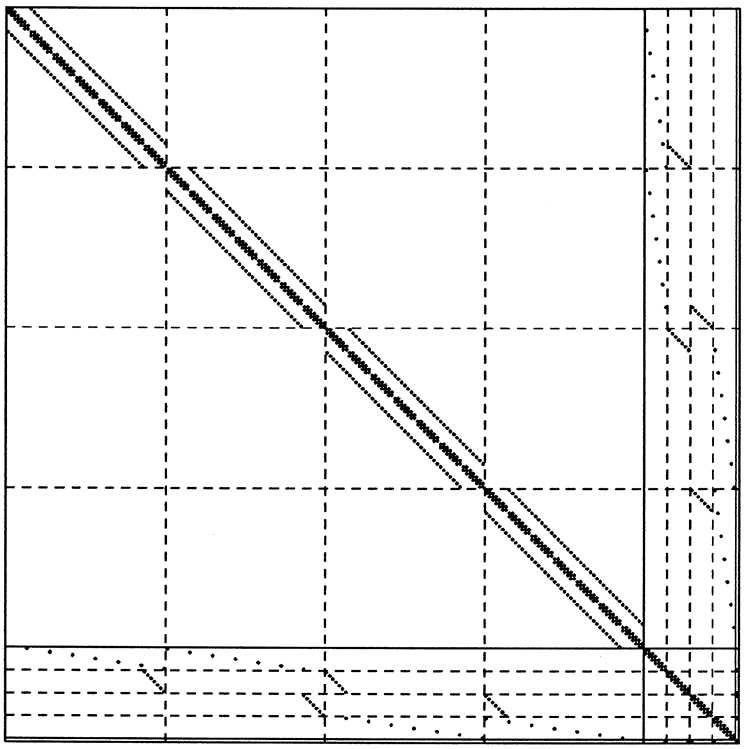

Figure 3: The partitioning of the matrix A_c for the 5-point Laplacian and four boxes arranged in a two by two manner. The dashed lines separate subdomains; the solid lines separate the interior unknowns from the interface unknowns and the cross point unknown.

cost is the need to guarantee safe access to shared data; this is often handled with barriers or more general critical regions. Sample hardware costs include bandwidth limits in shared resources such as memory buses and startup and transfer speeds in communication links. Perhaps the most subtle cost lies in algorithmic changes to "improve" parallelism; by choosing a poor algorithm over another, less parallel algorithm, artificially good parallel efficiencies can be found. We call a high parallel efficiency "artificial" if there exists an algorithm with lower parallel efficiency which nevertheless executes in less wall-clock time on a given number of processors. Of the many examples in the literature, one of the most dramatic is [5] on computing the forces in the n-body potential problem; the naive algorithm is almost perfectly parallel but substantially slower than the linear-in-n algorithm, which contains some reductions and hence some intrinsically serial computations.

We can identify the following costs in the domain decomposition algorithms that we are considering:

- Dot (inner) products. These involve a reduction, and hence at least $\log p$ time; in addition, there may be some critical sections (depending on the implementation).
- Matrix-vector products. These involve shared data, and hence may introduce some constraints on shared hardware resources.
- Preconditioner solves. These depend on the preconditioner chosen, and hence gives us the most freedom in trading off greater parallelism against superior algorithmic performance.

Note that the sharing of data is not random; most of the data sharing occurs between neighbors.

2.1. Message passing and Shared memory models

Two methods for achieving parallelism in computer hardware for MIMD machines are message passing and shared memory. In both of these, the software and hardware costs discussed above show up in the cost to access shared data. Each of these methods is optimized for a different domain, and these optimizations are reflected in the actual costs. In the following, to simplify the notation will will express all times in terms of the time to do a floating point operation. Further, we will drop constant factors from our estimates.

In a message passing machine, each processor has some local memory and a set of communication links to some (usually not all) other processors (called nodes). Each processor has access only to its own local memory. Communication of shared data is handled (usually by the programmer) by explicitly delivering data over the communication links. This takes time $s + rn$ for n words, where s is a startup time (latency) and r is the time to transfer a single word. This is good for local or nearest neighbor communication. For more global communication (such as a dot product), times depend on the interconnection network. For a hypercube, the global time is $(s + rn)\log p$; for a mesh, it is $(s + rn)\sqrt{p}$.

In a shared memory machine, each processor may directly access a shared global memory. Communication (access to) shared data is handled by simply reading the data. However, the actual implementation of this introduces a number of limits. For example, if the memory is on a common bus, then there is a limit to the number of processors that can simultaneously read from the shared memory. One way to model this cost is as $1 + p/\min(p, P)$ [7]. Here p is the number of processors and P is the maximum number of processors that may use the resource at one time.

In addition, the access to the shared data must be controlled; this can add additional costs in terms of barriers or critical sections. These can add additional terms which are proportional to $\log p$.

3. Complexity Estimates

We can estimate the computational complexity for these two models for several forms of domain decomposition. We note that these are rather rough estimates, good (because of their generality) for identifying trends. Additional estimates of the complexity of parallel domain decomposition may be found in [6].

3.1. Message Passing

In this case we can easily separate out the computation terms and the communication terms. For each part of the algorithm, we will place a computation term above the related communication term. In the formulas below, the constants in front of each term have been dropped for clarity.

In two dimensions, with n unknowns per side, we have for strips

Ax multiply	+	dot products	+	subdomain solves	+	interface solves
$\frac{n^2}{p}$	+	$\frac{n^2}{p} + \log p$	+	$\frac{n^2}{p}\log\frac{n}{p}$	+	$n \log n$ +
$s + rn$	+	$(s + r)\log p$	+	$s + rn$	+	$s + rn$

and for boxes, we have

Ax multiply	+	dot products	+	subdomain solves	+	interface solves	+	vertex solves
$\frac{n^2}{p}$	+	$\frac{n^2}{p} + \log p$	+	$\frac{n^2}{p}\log\frac{n}{\sqrt{p}}$	+	$\frac{n}{\sqrt{p}}\log\frac{n}{\sqrt{p}}$	+	$\sqrt{p}^{3/2}$ +
$s + r\frac{n}{\sqrt{p}}$	+	$(s + r)\log p$	+	$s + r\frac{n}{\sqrt{p}}$	+	$s + r\frac{n}{\sqrt{p}}$	+	$\sum_{i=1}^{\log p}(s + 2^{i-1}r)$

These costs are all per iteration. It is assumed that all neighbor-neighbor interactions can occur simultaneously. In the box case, the vertex coupling equations are solved by first exchanging

all of the vertex data with all the processors. Then each processor computes the entire vertex system. For the small systems considered, this is an efficient way to handle the vertex system. A more cooperative parallel solution approach would incur more communication startups and could actually take longer [11]. (A complexity comparison appears under #3 below.)

3.2. Shared Memory

In this case, a detailed formula depends on the specific design tradeoffs made in the hardware. The formula here applies to bus-oriented shared memory machines; a different formula would be needed for machines like the BBN Butterfly. These formulas are dominated by bandwidth limitations (the $\min(p, P)$ terms) and barrier or synchronization costs (the $\log p$ terms).

For strips, we have

$$\frac{n^2}{p}\left(a + \frac{bp}{\min(p, P_1)}\right) + 2\left(\frac{n^2}{p} + \log p\right) + 2\left(\frac{n^2}{p}\log\frac{n}{p}\right)\left(c + \frac{dp}{\min(p, P_2)}\right) + n\log n + 3\log p.$$

A similar formula holds for boxes. Here, a, b, c, d, P_1, and P_2 are all constants that depend on the particular hardware and implementation. P_i give a limit on the number of processors that can effectively share a hardware resource. The ratios a/b and c/d reflect the ratio of local work to use of the shared resource (such as memory banks or memory bus).

3.3. Implications

In domain decomposition algorithms, we can trade iteration count against work and parallel overhead. We will consider three representative tradeoffs:

1. No communication. This amounts to diagonal preconditioning. Call the number of iterations $I_{\text{decoupled}}$.
2. Local communication only. The FFT-implementable "$K^{1/2}$" preconditionings such as those in [2], where we can expect the iteration count to be proportional to p for strips and $\sqrt{p}$ for boxes. Call the number of iterations I_{local}.

   The cost of the "$K^{1/2}$" and all more implicit forms of preconditioning includes an extra subdomain solve to symmetrize the preconditioner and is roughly

   $$n\log n + \frac{n^2}{p}\log\frac{n}{p} + 2(s + rn)$$

   for a strip decomposition and a message passing system. The preconditioning complexity estimate is dominated by the subdomain solve terms, and it is thus roughly twice that of case 1. The communication costs are the same as for the matrix-vector product. Thus the local communication preconditioning is effective if

   $$2I_{\text{local}} \leq I_{\text{decoupled}}.$$

3. Global communication. In the case of box-wise decompositions, cross points occur at the intersections of the interfaces. The cross-points form a global linear system that is discussed in [2]. The iteration count is approximately independent of n as p varies; call it I_{global}.

   In this case, the additional costs over and above the symmetrization stage mentioned in #2 are those of communicating the entries in the cross-point problem and its solving it. For the case of a message passing system, we can do this in $p^2 + (s + r)\log p$ time if each processor solves the cross-point system and in time $p^{3/2} + p(s + r\sqrt{p})$ time if a parallel algorithm is used for the

solve, assuming a straightforward approach based on Banded Gaussian elimination on a ring. More sophisticated approaches for hypercubes which are $p + (s + r)\log p$ are known [3, 11].

Comparing this to the cost of the local computation of $(n^2/p)\log(n/\sqrt{p}) + (s + n)\log p$, the floating point work is negligible unless p is approximately equal to n, or higher. Assuming then that we use the local method, then the additional cost of the global communication makes the full cross-point method better whenever

$$I_{\text{global}}(2 - e) \leq I_{\text{local}},$$

where e is the parallel efficiency, equal to 1 minus the ratio of communication time to computation time.

For the shared memory case, if barriers and the dot product reduction are the dominant parallelism costs, the result is similar.

4. Experiments

The standard test problem considered was

$$\nabla^2 u = g$$

where $g = 32(x(1 - x) + y(1 - y))$ on the unit square. Though rather ideal, to minimize coefficient storage space, this problem has structural (from the graph-theoretic viewpoint) and spectral (from the operator-theoretic viewpoint) similarities to the generic self-adjoint elliptic problem and also to the non-self-adjoint problem in the limit of sufficiently high mesh refinement. The first set of experiments was conducted on an Encore Multimax 320 shared memory parallel computer with 18 processors; we used only 16, allowing the remaining 2 processors to handle various system functions. The experiments on the Encore Multimax were done in double precision. The results are shown in Tables 1 and 2. The Encore is a time-sharing machine, so these timings are accurate and have been reproduced only to about 10%. The computations were performed with no other users on the machine; however, various system programs (mailers, network daemons) used some resources. In addition, even a programmer who fully understands the dependencies in a given problem can not force each process to run on a different processor; operating system logic intervenes.

The tables show the iteration count I, an estimate of the condition number κ (estimated as in [9]), the time in seconds T, and the relative speedup s. The relative speedup is defined as the ratio of the time from the previous *column* with the time from the current column. The times do not include initial setup. While this slightly distorts the total time, it does allow the time per iteration to be determined by dividing the time by the iteration count.

The next set of experiments was run on a 64 node Intel iPSC/2-SX Hypercube, with 4 Megabytes of memory on each node and the SX floating point accelerator (a scalar floating point accelerator). All runs were in single precision to allow a large problem to fit in this memory space. The results are shown in Tables 3–6.

The programs in both of these cases were nearly the same. Only the code dealing with shared data was changed to use either messages or shared memory. (The Encore implementation is *not* a message passing implementation using the shared memory to simulate message; it is a "natural" shared memory code. In particular, the solution arrays are all shared. The Intel code is a "natural" message passing code. The solution arrays are all local (private).)

There are some differences between the iteration count results for the Encore and the Intel implementations. These differences seem to be due to the difference in floating point arithmetic on the two machines. Double precision was required on the Encore to get the fast Poisson solver we

$h^{-1}\backslash p$		1	2	4	8	16
16	I	1	3			
	κ	1.00	1.26			
	T	0.17	0.23			
	s		0.74			
32	I	1	2	5		
	κ	1.00	1.09	3.29		
	T	0.86	0.73	0.78		
	s		1.18	0.94		
64	I	1	2	4	8	
	κ	1.00	1.09	3.29	13.0	
	T	4.28	3.75	3.03	2.43	
	s		1.14	1.24	1.25	
128	I	1	2	4	8	16
	κ	1.00	1.09	3.29	13.0	51.9
	T	20.3	18.5	14.7	12.6	10.3
	s		1.10	1.26	1.17	1.22
256	I	1	2	4	8	16
	κ	1.00	1.09	3.29	13.0	51.9
	T	103	92.2	76.1	63.8	55.0
	s		1.12	1.21	1.19	1.16
512	I	1	2	4	8	16
	κ	1.00	1.09	3.29	13.0	51.9
	T	436	423	377	320	309
	s		1.03	1.12	1.18	1.04

Table 1: Results for strips on the Encore Multimax 320.

$h^{-1}\backslash p$		1	4	16
64	I	1	6	7
	κ	1.00	10.7	10.2
	T	4.28	5.43	1.53
	s		0.79	3.55
128	I	1	6	7
	κ	1.00	11.0	14.1
	T	20.3	27.2	6.9
	s		0.75	3.94
256	I	1	6	8
	κ	1.00	13.6	18.6
	T	103	130	38.3
	s		0.79	3.39
512	I	1	7	8
	κ	1.00	16.7	23.7
	T	436	708	179
	s		0.62	3.96

Table 2: Results for boxes on the Encore Multimax 320, using full vertex coupling.

$h^{-1}\backslash p$		1	2	4	8	16	32
32	I	1	2	5			
	κ	1.00	1.09	3.29			
	T	0.546	0.479	0.551			
	s		1.14	0.87			
64	I	1	2	4	8		
	κ	1.00	1.09	3.29	13.0		
	T	2.62	2.29	1.97	1.73		
	s		1.14	1.16	1.14		
128	I	1	2	4	8	19	
	κ	1.00	1.09	3.29	13.0	51.9	
	T	12.1	10.7	9.29	7.88	8.14	
	s		1.13	1.15	1.18	0.97	
256	I	1	2	4	8	19	41
	κ	1.00	1.09	3.29	13.0	51.9	207.5
	T	56	49.5	43.5	37.4	37.8	35.4
	s		1.13	1.14	1.16	0.99	1.07

Table 3: Results for strips on the Intel Hypercube.

$h^{-1}\backslash p$		1	4	16	64
32	I	1	5	7	
	κ	1.00	7.88	7.01	
	T	0.546	0.645	0.298	
	s		0.85	2.16	
64	I	1	6	7	6
	κ	1.00	10.7	10.2	7.16
	T	2.62	3.53	0.965	0.324
	s		0.74	3.66	2.98
128	I	1	6	7	7
	κ	1.00	11.04	14.1	10.5
	T	12.1	16.3	4.23	1.05
	s		0.74	3.85	4.03
256	I	1	6	8	8
	κ	1.00	13.57	18.6	14.8
	T	112	74.5	21.9	4.79
	s		1.50	3.40	4.57

Table 4: Results for boxes on the Intel Hypercube, using full vertex coupling.

used to work for $h = 1/512$. Single precision was required on the Intel in order to fit the problems in memory.

5. Comments

While the overall results for strips may seem poor, they actually represent very good speedup on a *per iteration* basis; it is the degraded iteration count for this type of iteration that suppresses the efficiency. We find, in accordance with the theory quoted in [8], that the condition numbers rise asymptotically quadratically in p, and hence that the number of preconditioned conjugate gradient iterations rises linearly in p, defeating the advantage of parallelism.

$h^{-1}\backslash p$		1	4	16	64
32	I	1	6	11	
	κ	1.00	12.1	25.3	
	T	0.546	0.76	0.41	
	s		0.72	1.33	
64	I	1	6	12	17
	κ	1.00	15.9	32.2	94.4
	T	2.62	3.52	1.59	0.674
	s		0.74	2.21	2.36
128	I	1	6	13	19
	κ	1.00	20.1	40.5	119.7
	T	12.1	16.3	7.79	2.57
	s		0.74	2.09	3.03
256	I	1	7	13	NA
	κ	1.00	24.8	49.6	NA
	T	112	87.0	35.5	NA
	s		1.29	2.46	NA

Table 5: Results for boxes on the Intel Hypercube, without vertex coupling but with interface preconditioning (neighbor coupling). The "NA" entries could not be computed on our iPSC/2.

Two of the full vertex coupling (global) results show superlinear relative speedup. This is a real effect, which derives from the superlinear growth in the cost of solving a single domain as a function of the number of points in the domain. This speedup is of course available to a single processor domain decomposition algorithm. In fact, the slight relative speedups seen for the strip decompositions are due almost entirely to this effect alone, since the number of iterations is proportional to the number of processors.

5.1. Comparison with the theory

In the case of the message passing results (Intel Hypercube), it is possible to fit the theoretical complexity estimate to the measured times. Taking only the highest order terms in the latency s and the arithmetic suggests a fit for the strip decomposition of

$$a_1 \frac{n^2}{p} + a_2 \frac{n^2}{p} \log \frac{n}{p} + a_3 + a_4 \log p. \tag{5.1}$$

We have ignored the r terms because $s \gg rn$ for given n on the Intel hypercube. A least squares fit to the data yields $a_1 = 0.00013$, $a_2 = 0.000090$, $a_3 = 0.000027$, and $a_4 = 0.0077$. These fits were computed by scaling the equations by the inverse of the time, making the least squares system $Ma = (1, 1, \ldots, 1)^T$. The residual $\|Ma - (1, \ldots, 1)^T\|_2$ is 0.044. The fit is shown in Figure 4. With these values (or directly from the data), the efficiency *per iteration* can be shown to be around 90% for the larger problems.

Some care is necessary in interpreting the graphs in Figure 4. In particular, the measure of efficiency used in the figure is relative to the single subdomain case for a single iteration. Because the cost of solving on a subdomain falls faster than linearly with increasing numbers of subdomains, this measure of efficiency will often be greater than one. An estimate of the parallel efficiency

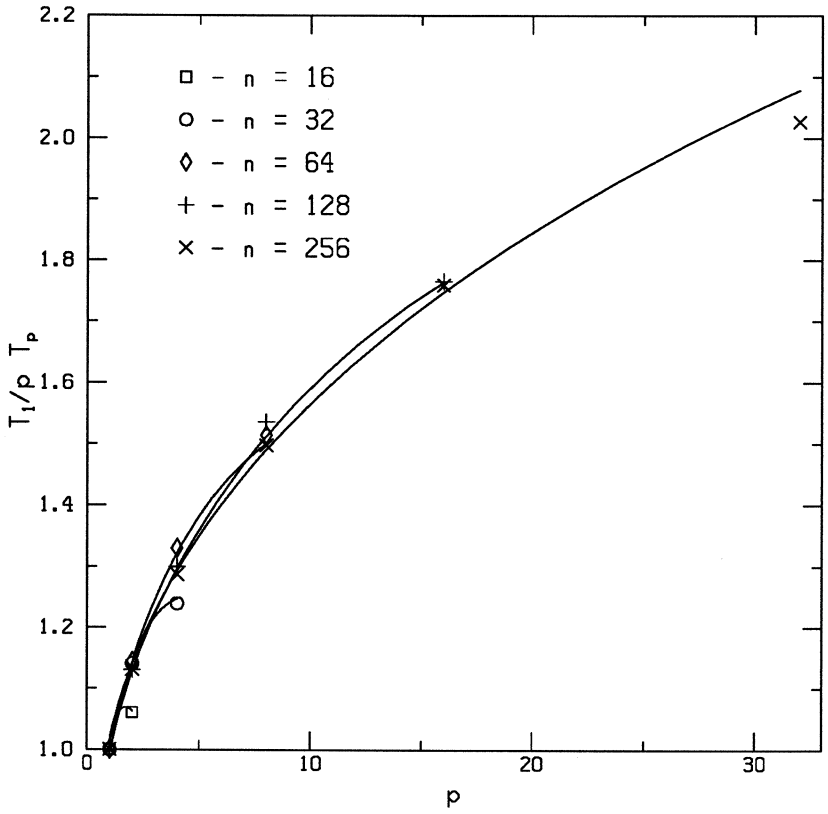

Figure 4: Data for iPSC/2 runs with strip decomposition. The fit is made using Equation (5.1). The y axis is the efficiency, defined as the time on a single processor, divided by the number of processors times the time on that many processors.

(computation time divided by total time) can be made from Equation (5.1):

$$E = \frac{a_1 \frac{n^2}{p} + a_2 \frac{n^2}{p} \log \frac{n}{p}}{a_1 \frac{n^2}{p} + a_2 \frac{n^2}{p} \log \frac{n}{p} + a_3 + a_4 \log p}. \tag{5.2}$$

Figure 5 shows the behavior of E with respect to p for a sample value of n.

For the box decompositions, the highest order terms in the complexity model are

$$a_1 \frac{n^2}{p} + a_2 \frac{n^2}{p} \log \frac{n}{p} + a_3(\sqrt{p} - 1)^3 + a_4 + a_5 \log p. \tag{5.3}$$

We have again ignored the r terms because of the high latency of the Intel hypercube. A least squares fit to the data yields $a_1 = 0.00012$, $a_2 = 0.000091$, $a_3 = 0.000014$, $a_4 = 0.0067$, and $a_5 = 0.0031$. The coefficients were computed in the same way as for strips, and the residual is 0.032. A graph of the data and the fit is shown in Figure 6. The a_3 term comes from the arithmetic complexity of solving the global cross-point system on a single processor using an already factored matrix. The efficiency *per iteration* is lower, because of the increased communication overhead, but is still above 70% for the larger problems. This makes the strip decomposition superior for moderate numbers of processors (where the iteration counts are similar).

To get a better idea for the individual overheads, the program for the Intel Hypercube was instrumented to provide data on the cost of each communication operation. This data showed

$h^{-1}\backslash p$		1	4	16	64
32	I	1	12	18	
	κ	1.00	29.1	49.2	
	T	0.546	0.827	0.382	
	s		0.66	2.16	
64	I	1	17	25	32
	κ	1.00	61.0	103.6	187.9
	T	2.62	5.45	1.81	0.741
	s		0.48	3.01	2.44
128	I	1	23	35	44
	κ	1.00	124.9	212.6	397.6
	T	12.1	29.6	11.5	3.28
	s		0.41	2.57	3.51
256	I	1	27	48	62
	κ	1.00	254.9	432.4	818.0
	T	112	182	71.4	20.5
	s		0.62	2.55	3.48

Table 6: Results for boxes on the Intel Hypercube, using diagonal blocks only (no coupling.)

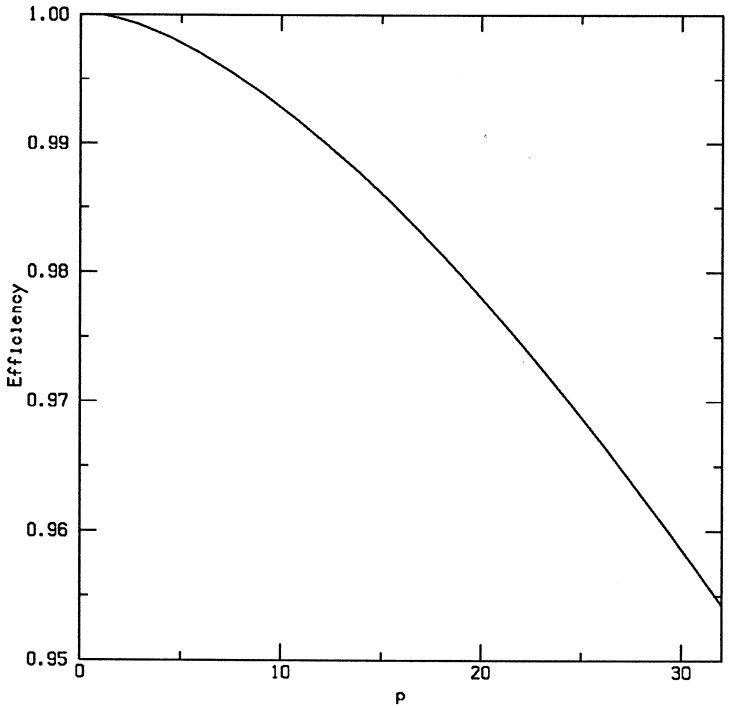

Figure 5: Estimated parallel efficiency for the strips case on the iPSC/2, using Equation (5.2), $n = 255$, and the parameters in the text.

that the expense of the global communication used to form the vertex coupling system was always smaller than the communication to form the dot-products, and smaller in all but one case than the nearest-neighbor communication used in forming the matrix-vector product Ax. This result is consistent with the known communication behavior of the Intel Hypercube. The two dot products

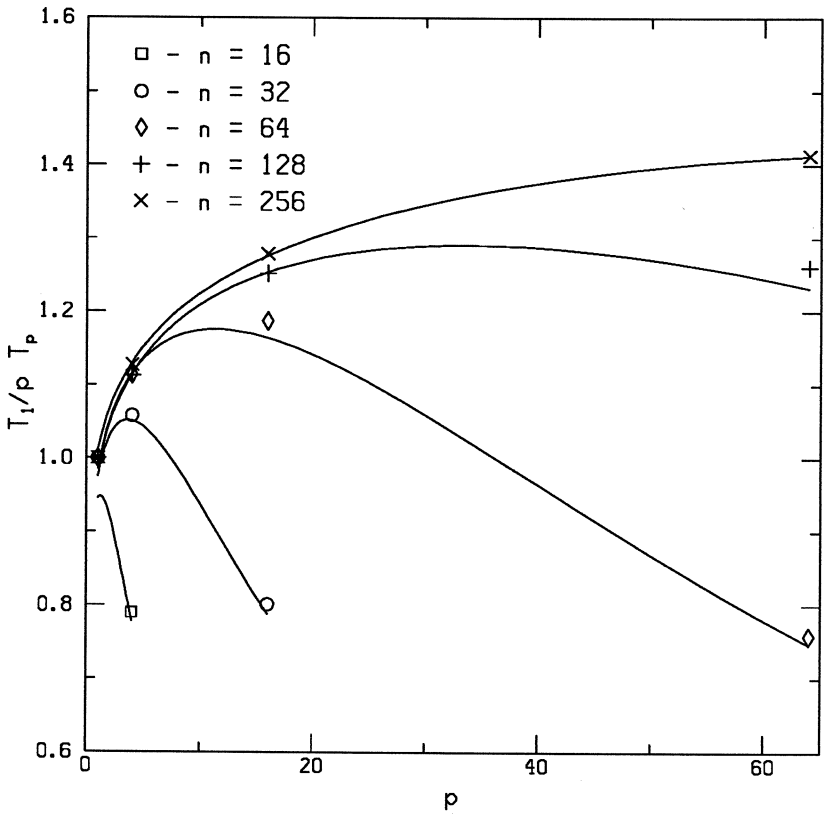

Figure 6: Data for iPSC/2 runs with full vertex coupling. The fit is made using Equation (5.3). The y axis is the efficiency, defined as the time on a single processor, divided by the number of processors times the time on that many processors.

done in each iteration involve a time $2(s+r)\log p$ while the global exchange involves (roughly) $s\log p + pr$; since $s \gg r$, the global exchanges take less time. In the case of the nearest-neighbor communications, the comparison is with $4(s + rn/\sqrt{p})$. The larger amount of data moved in the neighbor computation makes it often more expensive than the global communication for the vertex coupling. Further, when n is small, we would expect the neighbor communication to be faster than the global communication, and this is exactly what is observed. A different balance of s and r, or significantly more processors would of course change these results. For example, on a truely massively parallel machine with thousands of processors, the neighbor communication would be smaller than either the dot-product or global communication.

This leads to the results in Tables 5 and 6, where a simpler communication strategy has been traded against larger iteration counts.

For the shared memory machine, the complexity estimates are harder to demonstrate. In part, this is due to the design of the shared memory machines; the number of processors is deliberately limited to roughly what the hardware (i.e., memory bus) can support. The dominant effect is usually load balancing or the intrinsically serial parts of the computation (synchronization points and dot products).

The results for the Intel Hypercube are summarized in Table 7 which gives the optimal choice of domain decomposition algorithm (from among those considered) for various values of p and h for the Intel Hypercube. As either the number of processors or the number of mesh points increase, the global or full vertex coupling algorithm becomes more efficient, despite its additional communication

$p \backslash h^{-1}$	16	32	64	128	256
4	Decoupled	Global	Local	Local	Global
16		Global	Global	Global	Global
64			Global	Global	Global

Table 7: Optimal choice of algorithm for the given problem and implementation on the Intel Hypercube, from the choices of decoupled block diagonal, locally coupled interfaces, and full vertex coupling preconditioners for the box decomposition.

demands. These results emphasize the importance of considering a wide range of decompositions and problem sizes. For example, the results for $p = 4$ can give a misleading picture of the usefulness of both the fully decoupled and the locally coupled interface preconditioners. In fact, the "local" preconditioning wins by an extremely narrow margin in the two cases in which it beats out "global". Our results show that, even for relatively small problems, the asymptotically superior performance (in iteration count) of the full vertex coupling preconditioners more than compensates for the the additional cost of the communication. This is true even for a machine such as the Intel iPSC/2 Hypercube, with relatively slow communication and global communication that is proportional to $\log p$. The older iPSC/1 Hypercube had an identical table, which is not surprising, given that the ratio of communication speed to computation speed for the two machines is similar.

References

[1] C. Borgers, *The Neumann-Dirichlet Domain Decomposition Method with Inexact Solvers on the Subdomains,* 1989. Numer. Math. (to appear).

[2] J. H. Bramble, J. E. Pasciak, and A. H. Schatz, *The Construction of Preconditioners for Elliptic Problems by Substructuring, I,* Mathematics of Computation, 47 July (1986), pp. 103–134.

[3] T. F. Chan, Y. Saad, and M. H. Schultz, *Solving Elliptic Partial Differential Equations on the Hypercube Multiprocessor,* Technical Report YALE/DCS/RR-373, Yale University, Department of Computer Science, March 1985.

[4] M. Dryja, *A Finite Element - Capacitance Method for Elliptic Problems on Regions Partitioned into Subregions,* Numer. Math., 44 (1984), pp. 153–168.

[5] L. Greengard and W. D. Gropp, A Parallel Version of the Fast Multipole Method, *Parallel Processing for Scientific Computing,* SIAM, 1989, pp. 213–222.

[6] W. D. Gropp and D. E. Keyes, *Complexity of Parallel Implementation of Domain Decomposition Techniques for Elliptic Partial Differential Equations,* SIAM Journal on Scientific and Statistical Computing, 9/2 (1988), pp. 312–326.

[7] H. F. Jordan, *Interpreting Parallel Processor Performance Measurements,* SIAM Journal on Scientific and Statistical Computing, 8/2 (1987), pp. s220–s226.

[8] D. E. Keyes and W. D. Gropp, *A Comparison of Domain Decomposition Techniques for Elliptic Partial Differential Equations and their Parallel Implementation,* SIAM Journal on Scientific and Statistical Computing, 8/2 (1987), pp. s166–s202.

[9] D. P. O'Leary and O. Widlund, *Capacitance Matrix Methods for the Helmholtz Equation on General Three-dimensional Regions,* MATHCOMP, 33 (1979), pp. 849–879.

[10] J. S. Przemieniecki, *Matrix Structural Analysis of Substructures,* AIAA J., 1 (1963), pp. 138–147.

[11] Y. Saad and M. Schultz, *Parallel direct methods for solving banded linear systems,* Technical Report YALE/DCS/RR-387, Yale University, Department of Computer Science, August 1985.

[12] H. A. Schwarz, *Gesammelte mathematische Abhandlungen,* Berlin, Springer, 2 (1890), pp. 133-143. (First published in 1870).

[13] O. B. Widlund, *Iterative Substructuring Methods: the General Elliptic Case,* Technical Report 260, Courant Institute of Mathematical Sciences, NYU, November 1986.