CHAPTER 22

Domain Decomposition on Parallel Computers

William D. Gropp*
David E. Keyes†

Abstract. We consider the application of domain decomposition techniques for the solution of sparse linear systems on parallel computers. We consider two representative types of MIMD parallel computer; a message passing and a shared memory architecture. For each we develop complexity estimates and compare these against actual computations. Various tradeoffs in parallel computation costs are discussed. Our complexity estimates are tested for a variety of methods, decompositions, and problem sizes. Results from both an Intel iPSC Hypercube and an Encore Multimax are presented.

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 several 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 also look at the tradeoffs between various forms of preconditioning, categorized 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

*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.

parallel processor is optimized for a different interprocessor communication pattern, and we consider the effects of these optimizations on domain decomposition.

We consider only partitioned matrix methods applied to preconditioned conjugate gradient techniques such as those reviewed in [5]. Briefly, the matrix is partitioned into subdomains connected by small (lower dimension) interface regions. For a single domain decomposed into two subdomains (1 and 2) connected by an interface (3), the partitioned matrix would look like

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

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 bewteen the subdomains and the interfaces. We are interested in various preconditioners for A, 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 [1], which requires the solution of a linear system for the cross points. This method involves only low bandwidth global communication (that is, the size of the messages scales with the number of processors); we will not consider any method which uses high bandwidth communication (on the order of the size of the problem).

We develop a complexity model for both types of parallel computer which 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 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 involves 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 software cost is the need to guarantee safe access to shared data; this is often handled with barriers or more general critical regions. Example 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 with better parallelism than another, less parallel algorithm, artificially good results can be found. An example is computing the forces in the n-body 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) [3].

We can identify these costs with 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 include 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 constants (like 2) 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 only has access 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 start up 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 $1 + p/\min(p, P)$ [4]. 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.

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.

For strips, we have

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

These costs are all per iteration. It is assumed that all neighbor-neighbor interactions can occur simultaneously.

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 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

For domain decomposition, 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 "$K^{1/2}$" preconditionings such as those in [1], 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 "$K^{1/2}$" 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 is dominated by the extra solve for the "harmonic" component; 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 [1]. The iteration count is roughly constant; call it I_{global}.

In this case, the additional cost is that of a cross-point solve and the communication of the entries in the cross-point problem and its solution. 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 gaussian elimination on a ring. More sophisticated approaches for hypercubes which are $p + (s + r)\log p$ are known [2].

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 $n \approx p$ (when the problem is essentially all cross-points). 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. 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 1–2. We note that the Encore is a time sharing machine, so these times are accurate to only about 10%. The tables show the iteration count I, an estimate of the condition number κ, 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, including the cost of the initial matrix vector multiply and preconditioner solve. 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 actual computations were performed with no other users on the machine; however, various system programs (mailers, network daemons) used some resources. In addition, the programmer can not force each process to run on a different processor.

The next set of experiments was run on a 64 node Intel Hypercube, with 4.5 Megabytes of memory on each node. 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 (in particular, the Encore implementation is *not* a message passing implementation using the shared memory to simulate message; it is a "natural" shared memory code. The Intel code is a "natural" message passing code.)

There are some differences between the 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

$h^{-1}\backslash p$		1	2	4	8	16
16	I	1	3			
	κ	1.00	1.26			
	T	0.2	0.22			
	s		0.91			
32	I	1	2	5		
	κ	1.00	1.09	3.29		
	T	0.97	0.80	0.78		
	s		1.21	1.03		
64	I	1	2	4	8	
	κ	1.00	1.09	3.29	13.0	
	T	4.95	4.13	3.33	2.63	
	s		1.2	1.24	1.27	
128	I	1	2	4	8	16
	κ	1.00	1.09	3.29	13.0	51.9
	T	24.3	20.0	16.9	14.2	12.1
	s		1.22	1.18	1.19	1.17
256	I	1	2	4	8	16
	κ	1.00	1.09	3.29	13.0	51.9
	T	108	105	84.6	71.9	63.7
	s		1.03	1.24	1.18	1.13
512	I	1	2	4	8	16
	κ	1.00	1.09	3.29	13.0	51.9
	T	513	481	420	353	327
	s		1.07	1.15	1.19	1.08

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	4.95	6.17	1.55
	s		0.76	3.98
128	I	1	6	7
	κ	1.00	11.0	14.1
	T	24.3	31.2	7.7
	s		0.78	4.05
256	I	1	6	8
	κ	1.00	13.6	18.6
	T	108	143	43.4
	s		0.76	3.29
512	I	1	7	8
	κ	1.00	16.7	23.7
	T	513	864	205
	s		0.59	4.21

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	6.08	5.14	5.22			
	s		1.18	0.98			
64	I	1	2	4	8		
	κ	1.00	1.09	3.29	13.0		
	T	29.9	25.9	21.2	17.3		
	s		1.15	1.22	1.23		
128	I	1	2	4	8	18	
	κ	1.00	1.09	3.29	13.0	51.9	
	T	142	125	105	85.9	78.9	
	s		1.13	1.19	1.22	1.09	
256	I	1	2	4	8	18	41
	κ	1.00	1.09	3.29	13.0	51.9	207.5
	T	652	588	508	425	390	361
	s		1.11	1.16	1.20	1.09	1.08

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	6.08	6.93	3.29	
	s		0.88	2.11	
64	I	1	6	7	6
	κ	1.00	10.7	10.2	7.16
	T	29.9	40.4	10.9	4.5
	s		0.74	3.71	2.42
128	I	1	6	7	7
	κ	1.00	11.04	14.1	10.5
	T	142	195	48.4	12.6
	s		0.73	4.03	3.84
256	I	1	6	8	8
	κ	1.00	13.57	18.6	14.5
	T	652	925	262	57.7
	s		0.70	3.53	4.54

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

solver we used to work for $h = 1/512$. Single precision was required on the Intel 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.

A number 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 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.

$h^{-1}\backslash p$		1	4	16	64
32	I	1	6	11	
	κ	1.00	12.1	25.3	
	T	6.08	8.15	4.34	
	s		0.75	1.88	
64	I	1	6	12	17
	κ	1.00	15.9	32.2	94.4
	T	29.9	40.2	17.4	8.44
	s		0.74	2.31	2.06
128	I	1	6	13	19
	κ	1.00	20.1	40.5	119.7
	T	142	195	88.9	30.1
	s		0.73	2.19	2.95
256	I	1	7	13	33
	κ	1.00	24.8	49.6	794.8
	T	652	1080	425	231
	s		0.60	2.54	1.84

Table 5: Results for boxes on the Intel Hypercube, without vertex coupling but with interfaces.

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 computation suggests a fit for the strip decomposition of

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

We have ignored the r terms because $s \gg rn$ for given n on the Intel hypercube. To avoid any problems with small n/p (such as not being in the asymptotic regime for the FFT and fast Poisson solvers), we eliminated the data with $1/(hp) = 8$. A least squares fit to the data yields $a_1 = 0.00048$, $a_2 = 0.0012$, $a_3 = 0.027$, and $a_4 = 0.030$. The relative residual is 0.038. With these values (or directly from the data), the efficiency *per iteration* can be shown to be around 90% for the larger problems.

The data for the box decompositions is harder to fit (in part because our model does not include some implementation effects on the Intel and because the communication terms are small). However, the formula

$$0.00021\frac{n^2}{p} + 0.0013\frac{n^2}{p}\log\frac{n}{\sqrt{p}} + 0.0023\frac{n}{\sqrt{p}} + 0.072\log p$$

is a reasonable fit with relative residual 0.092. The efficiency *per iteration* is lower, because of the increase 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).

This leads to the results in Tables 5 and 6, where a simpler communication strategy has been traded against larger iteration counts. The results are summarized in Table 7, which shows the optimal choice of preconditioners for our implementation on the Hypercube. For significant parallelism, the globally coupled preconditioner prevails in spite of its higher communication overhead.

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

$h^{-1}\backslash p$		1	4	16	64
32	I	1	12	18	
	κ	1.00	29.1	49.2	
	T	6.08	8.62	3.95	
	s		0.71	2.18	
64	I	1	17	25	32
	κ	1.00	61.0	103.6	187.9
	T	29.9	60.6	20.1	10.1
	s		0.49	3.01	1.99
128	I	1	23	35	44
	κ	1.00	124.9	212.6	397.6
	T	142	395	128	39.7
	s		0.36	3.09	3.22
256	I	1	27	48	210
	κ	1.00	252.9	430.8	23480.
	T	652	2170	829	789
	s		0.30	2.62	1.05

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

$p\backslash h^{-1}$	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.

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).

References

[1] 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.

[2] 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.

[3] L. Greengard and W. D. Gropp, *A Parallel Version of the Fast Multipole Method,* To appear in the proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing, December 1–4, 1987.

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

[5] 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.