CHAPTER 29

Parallel Computing and Domain Decomposition

William Gropp*

Abstract. Domain decomposition techniques appear a natural way to make good use of parallel computers. In particular, these techniques divide a computation into a local part, which may be done without any interprocessor communication, and a part that involves communication between neighboring and distant processors.

This paper discusses some of the issues in designing and implementing a parallel domain decomposition algorithm. A framework for evaluating the cost of parallelism is introduced and applied to answering questions such as which and how many processors should solve global problems and what impact load balancing has on the choice of domain decomposition algorithm. The sources of performance bottlenecks are discussed. This analysis suggests that domain decomposition techniques will be effective on high-performance parallel processors and on networks of workstations.

1. Introduction. Domain decomposition methods have become very popular in recent years. Of the many advantageous features claimed for these methods, the ability to be used on parallel computers is one of the most cited and least examined. In this paper, we discuss some of the issues in developing an efficient parallel domain decomposition algorithm and the reasons that domain decomposition is, in fact, a good approach for parallel computers. The paper starts by describing the structure of domain decomposition methods as it applies to parallel computing. Then, the realities of parallel computing are discussed, and a mathematical model for the additional terms in a time-complexity analysis of a parallel algorithm is described. A key feature of this model is its *two-level* memory structure. This two-level structure is shown to reflect the structure of many domain decomposition algorithms. Finally, the overheads and bottlenecks in these algorithms are discussed. For the reader who wishes the punchline in advance, there are two major points to this paper. First, domain decomposition algorithms with their two- (or three-) level structure efficiently match the two- (or three-) level structure of actual high-performance parallel computers. Second, the costs of interprocessor communication and load balancing are important and

* Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439. *Email:* groppmcs.anl.gov. This work was supported by the Applied Mathematical Sciences subprogram of the Office of Energy Research, U.S. Department of Energy, under Contract W-31-109-Eng-38.

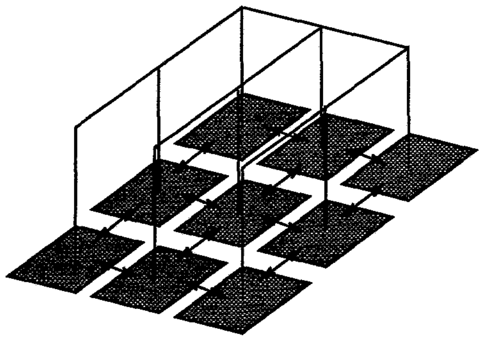

FIG. 1. *Sample communication structure of domain decomposition algorithms*

can effectively guide the design and implementation of domain decomposition algorithms.

For a more detailed examination of the time complexities of domain decomposition algorithms, see [6, 15]. For a sampling of results about parallel domain decomposition methods, see [1, 3, 7, 8, 9, 10, 11, 14, 16].

2. Structure of Parallel Domain Decomposition Methods.

Domain decomposition methods seem ideally suited for parallel computers. In their simplest form, each domain may be solved on a separate processor, yielding an apparently perfectly parallel algorithm. For a number of reasons, this is an illusion. Most important, any domain decomposition algorithm involves some communication or coordination between the computations on each domain; a typical case is shown in Figure 1.

The most important fact to notice in Figure 1 is that there are three levels: the domains themselves (shaded in gray), communication with neighboring domains (double headed arrows), and global communication (the wire frame connecting all of the domains). These levels correspond to the operations in a domain decomposition algorithm, which usually involves three kinds of computation. These computations are an interior solve, an evaluation of the matrix-vector product, and the computation of a small number of quantities over the whole domain (e.g., dot products, cross-point solutions). The efficiency of a domain decomposition method depends on how well these levels can be mapped onto a parallel computer. As we shall show, from the point of view of parallelism, it is the global communication part that has the largest impact on the available parallelism. Determining the best way to handle this part is the focus of this paper. No choice of method will eliminate some loss of parallel efficiency at this step.

These observations on communication requirements apply to both implicit and explicit algorithms, and to "asymptotic" domain decomposition (where the domains are chosen based on the local properties of the equations). For problems without a global part (the "wire frame" in Figure 1), load balancing is the biggest concern.

3. Realities of Parallel Computing.

In a perfect world, parallel computers would be as easy to use as uniprocessor computers. Unfortunately, parallel computers represent a series of design compromises. Of course, *any* parallelism in the processor is an admission that a single processor could not be made that met desired performance or cost requirements. There are two principal places where parallelism is introduced in order to improve performance: processors and memory. The kind of connection between the processors and the memory is the primary basis for distinguishing among types of parallel processor. Common forms of processor-to-memory connection are shown in Figures 2 and 3. These two forms are distinguished by how much of the total memory each processor can access. In a shared-memory parallel computer, all of the memory is accessible to every processor. Communication between processors is carried out by writing and reading shared memory. In a distributed-memory parallel computer, only the memory attached to a processor is accessible by that processor. Communication between processors is carried out by sending messages from one processor to another. In both kinds of parallel computer, it takes more time to access a memory location that is "far away." The penalty for accessing faraway memory can be large; actual values range from a factor of 3 (on a network-based shared-memory machine) to several orders of magnitude (on a distributed-memory machine). Thus, it is important to manage the use of memory in order to achieve good efficiency. The best approach is to understand the sources and relative sizes of the costs; this can be done in a relatively simple way by modeling the cost of communicating information between processors.

A number of metrics for measuring the performance of a parallel algorithm have been proposed. The simplist are efficiency, defined as $E_p = T_1/(pT_p)$, and speedup, defined as $S_p = T_1/T_p$. Here, T_1 is the time to execute an algorithm on a single processor, and T_p is the time to execute the *same* algorithm on p processors. At the very least, we wish $\partial S_p/\partial p > 0$ (otherwise, adding processors slows down the computation). A *perfectly parallel algorithm* has $E = 1$. It is important to note that since T_1 and T_p refer to the same algorithm, neither speedup nor efficiency is a reliable indicator of quality. For example, by picking a poor, computation-intensive algorithm, the efficiency can be made very close to one. Perhaps the best measure of efficiency would have T_1 refer to the best algorithm on a single processor. Unfortunately, it is difficult to get any agreement on what the best algorithm for any problem is.

3.1. The Important Parameters. In analyzing the time complexity of a parallel program, there are two new major costs. One is communication, and the other is load balancing. For most distributed-memory computers, communication costs may be modeled as

$$s + rn,$$

where s is the start-up time, r is the time to transfer a single word, and n is the number of words. For a bus-oriented shared-memory computer, the cost is roughly

$$\frac{rn}{\min(k,p)},$$

where k is the maximum number of simultaneous requests on the bus [13].

It is convenient to express the times s and r relative to the cost to do a single floating-point operation (f), and we will do so throughout this paper. For many (but not all) distributed-memory parallel computers,

$$s \gg r > f.$$

Note that this gives a very clear *two-level* structure to memory. Local memory may be accessed quickly (typically with time approximately f). Global memory, that is, memory on another processor, can be accessed only at much greater cost ($s \gg f$).

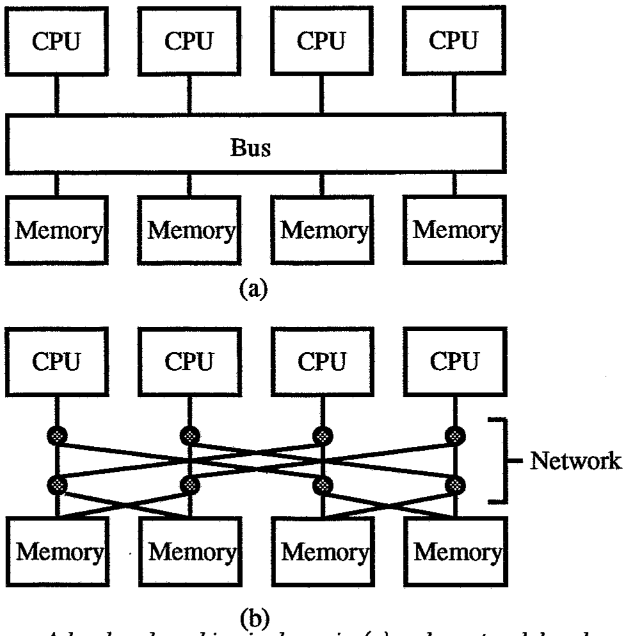

FIG. 2. *Shared memory. A bus-based machine is shown in (a) and a network-based machine is shown in (b).*

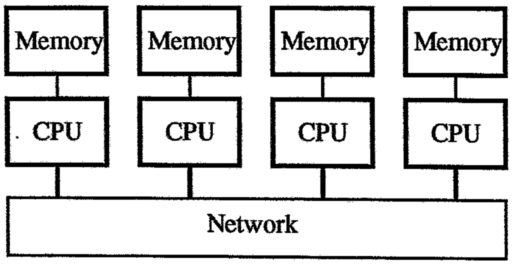

FIG. 3. *Distributed memory*

3.2. Load Imbalances. In any parallel computation where the parallel processors must coordinate their work, it is difficult to keep all processors busy all of the time. Often, some processors have to wait for others to finish their work before they can proceed with their next task. For example, consider the case where $p-1$ processors have work W_1 and the last has work $W_2 > W_1$. The *best* possible speedup is then

$$\begin{aligned} S_p &= \frac{(p-1)W_1 + W_2}{W_2} \\ &= 1 + (p-1)\frac{W_1}{W_2}. \end{aligned}$$

Because the smallest practical unit of work is *not* a single arithmetic operation, but a "module" or "subroutine," it becomes increasingly difficult to keep the range of work small as the scale of parallelism increases.

4. Unavoidable Overhead. It is natural to ask whether an efficient, perfectly parallel algorithm for solving partial differential equations (PDEs) numerically exist. Unfortunately, there is no such algorithm. Worley [17] has shown that perfectly parallel algorithms for PDEs do not exist and that, for a given accuracy, there is a lower bound on the time it will take to achieve this accuracy. It is important to note that this is not a bound on speedup: by adding more purely local work, speedup can be made arbitrarily close to 1.

In practice, this overhead shows up as local communication, global communication, and coordination. Each of these may be traded off against the others, though with potentially great cost. For example, by using only local communication such as in a relaxation algorithm, the cost per iteration may be kept independent of the number of processors, but the algorithm will require more iterations to compute the solution. Asynchronous algorithms do away with the coordination cost, but again at a penalty in iteration count. Thus the true goal in the design of a parallel algorithm for solving a PDE is to achieve the most efficient combination of these overheads, where efficiency is in terms of minimum elapsed time. Domain decomposition algorithms are good candidates for efficient parallel algorithms because their structure matches that of parallel computers.

4.1. The cross-point problem. The global cross-point problem is source of both the algorithmic efficiency of many domain decomposition algorithms and the parallel inefficiency. In this section, the behavior of various approaches to solving the global cross-point problem is analyzed.

4.2. Duplicate work. One sometimes surprising feature of parallel algorithms is the fact that it is sometimes more efficient for many processors to compute the same result (a redundant computation) than to have one processor compute and distribute that same result. It turns out that the solution of the cross-point system in domain decomposition algorithms can be such a case. In fact, many of the issues in analyzing a parallel algorithm are illustrated by finding an answer to the question, "How should the cross-point grid system be solved?"

There are, of course, an great many ways to solve the cross-point system. Some obvious possibilities are

1. in parallel,
2. on one processor,
3. on all processors separately, and
4. on a subset of processors.

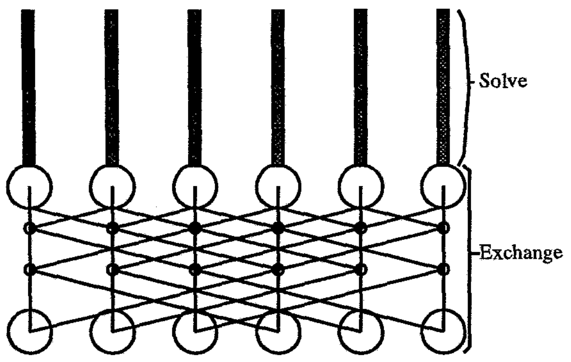

FIG. 4. *On all processors. A parallel solve looks like this, except the solve and exchange steps are intermixed.*

Naturally, the answer to the question will depend not only on the details of the parallel computer but also on the particular choice of numerical algorithm. To cover the most ground, we shall consider several different numerical algorithms for solving the linear system. Also, for concreteness we shall consider an $n \times n$ grid of cross-points and a distributed-memory parallel computer with p processors. Note that since the cross-point system will probably be small relative to the size of the parallel processor (probably $n^2 = p$), results that are for $n/p \to \infty$ may be misleading.

4.3. All or one. Let us first look at the question of solving the cross-point system on all the processors in parallel, or solving the problem in serial on one (or more) processors. The critical point here is that there is a tension between communication and computation. We can look at the minimum costs to communicate the data first, and then compare with the computational costs.

If each processor solves the cross-point system, the only communication is the collection of the right-hand-sides. This is illustrated in Figure 4. The communication takes time

$$T_{exch} = (s + n^2 r) \log p.$$

If we use banded Gaussian elimination on the each processor to solve the system of equations, the total cost is

$$T_{serial} \approx n^3 + (s + n^2 r) \log p.$$

Now, consider using parallel banded Gaussian elimination. The cost for this is

$$T_{parallel} > n^3/p + 2(p-1)\left(s + \frac{n^2}{p} r\right),$$

where important load-imbalance effects have been ignored. (See [4, 12] and references there for a detailed discussion of the time complexity of parallel banded Gaussian elimlination. The time here is for only the solve and does not include the cost of factorization.)

For the parallel solve to be faster, we need

$$n^3/p + 2(p-1)\left(s + \frac{n^2}{p} r\right) < n^3 + (s + n^2 r) \log p.$$

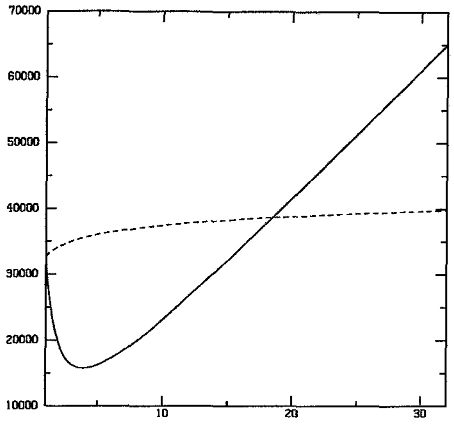

FIG. 5. *Comparison of serial and parallel banded Gaussian elimination. The solid line is the time for the parallel algorithm; the dashed line is the time for the uni-processor algorithm on each processor.*

The high communication cost of the parallel solver will often overwhelm the reduced computation. For example, consider the case of one cross-point on each processor: $n^2 = p$. Then we would need

$$n + 2(n^2 - 1)(s + r) < n^3 + (s + n^2 r)2 \log n.$$

For parallel computers with large s, this is very roughly

$$2n^2 s < n^3 + 2(s + n^2 r) \log n.$$

This inequality will not be satisfied if both $n^2 s > n^3$ and $n^2 s > 2(s + n^2 r) \log n$. The first is just $s > n = \sqrt{p}$, which is true for most distributed-memory computers. The second is roughly $s > r \log p$, which is again true for most distributed-memory computers. Thus, for this relatively small system of equations, it is slower to use many processors than to use one processor.

If a method other than banded Gaussian elimination is used, then the analysis must be repeated. Note, however, that a sufficient condition for the solve on a single processor to be more efficient is for

$$T_{serial} < T_{parallel\ communication}.$$

For example, if multigrid (V cycle) is used instead of banded Gaussian elimination, each half-cycle requires log(n) communication steps, with the ith step sending data a distance of 2^i (see [2] for a discussion of the time complexity of parallel multigrid). If there are I cycles, the time will be roughly

$$T_{parallel\ communication} = I(\sum_{i=0}^{\log n} (s + 2^i r)) = I(s \log n + nr),$$

and the serial time will be

$$T_{serial} = In^2 + (s + n^2 r) \log n.$$

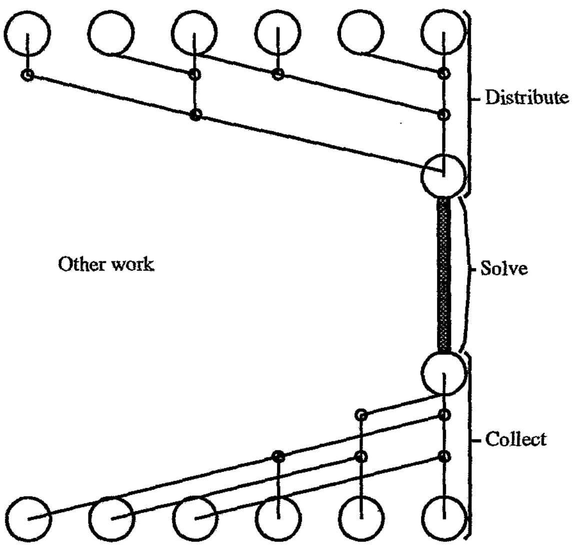

FIG. 6. *On one processor*

Comparing these, we see that for the parallel version to be faster, we need

$$I(s \log n + nr) < In^2 + (s + n^2 r) \log n.$$

For $p = n^2$, this reduces to

$$I(s \log n + nr) < In^2 + (s + n^2 r) \log n,$$

or roughly

$$Is \log n < In^2$$

or

$$s < \frac{n^2}{\log n} = \frac{2p}{\log p}.$$

While this is a less severe constraint than Equation 4.3, it is still a stringent requirement, and one that most distributed memory parallel computers do not meet.

Thus it can be cheaper to do duplicate work. (An intermediate choice is suggested by Figure 5—use clusters of $p_0 < p$ processors.) The problem here is the communication time; a method requiring less computation may not require less communication, thus reducing the method's parallel efficiency. Another way to look at the situation is that there is not enough data per communication. A similar computation can be carried out to decide whether to *factor* the problem, when using Gaussian elimination, on all or some of the processors.

4.4. Overlapped Work. Once we have decided that it is better to solve each cross-point problem on a single processor, we must ask whether it is better to solve on a single processor and distribute the results to the other processors, or solve the identical problem on all of the processors. Intuitively, we might expect to be able to accomplish some other "useful" work on the other processors if we solve the cross-point problem on a single processor. This is illustrated in Figure 6. However, this requires us to distribute the solution that is computed on the single processor, and we shall see that this can be a significant cost. The cost has two components: balancing the work and sharing the results of the cross-point grid solution.

An example of a method that allows the overlap of the solution of the cross-point problem and other work are the additive methods, such as the additive Schwarz method [5]. These methods allow *all* of the subproblems to be solved in parallel, seemingly avoiding any coordination overhead. However, different phases of this computation have differing loads:

- Solves—one processor has the cross-point system in addition to local solves.
- Matrix multiply, dot products, updates—work is proportional to the number of mesh points.

While the differing loads presented by these two phases are an important consideration (see Figure 7), we shall analyze only the additional communication cost incurred by having only one processor solve the cross-point system.

4.5. Distributing the solution. Let the cross-point system be solved with an optimal method:

$$T_{solve} = cn^2.$$

The total cost to solve the cross-point problem on one processor and distribute is (summed over all processors)

$$2pT_{coll} + T_{solve}.$$

If each processor solves the same cross-point problem, the total time is

$$p(T_{coll} + T_{solve}).$$

Less total time is consumed in solving the cross-point problem if

$$T_{coll} < \frac{p-1}{p} T_{solve}.$$

This is true if

$$\begin{aligned} (s + rn^2)\log p &< \left(\frac{p-1}{p}\right) cn^2 \\ r &< \left(\frac{p-1}{p}\right)\frac{c}{\log p} - \frac{s}{n^2} \\ &< \frac{c}{\log p} - \frac{s}{n^2}. \end{aligned}$$

Since s/n^2 is likely to be small, this depends critically on c. For fast enough solvers, the cost of moving the data around can exceed the cost of solving the cross-point system (particularly if an approximate solution can be used). Thus, there may be no savings in overlapping the work of the cross-point system with other work.

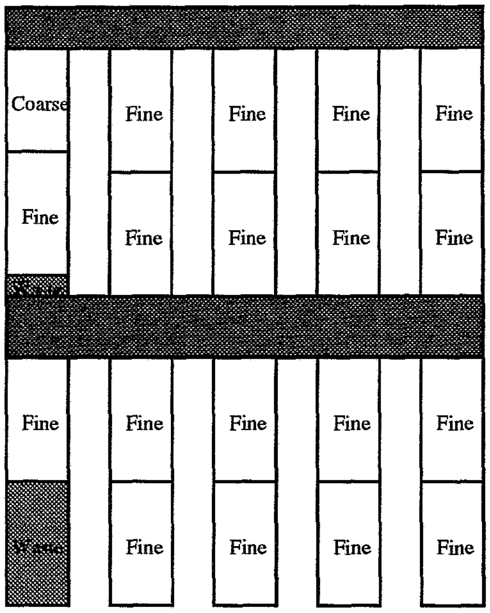

FIG. 7. *Sample load distribution when one processor solves the cross-point problem. Dark gray indicates communication (an exchange of data), light gray idle or wasted time, "coarse" the solution of the cross-point problem, and "fine" operations on the subdomains.*

It is also important to note that even if the above analysis suggests that it is best to solve the cross-point system on one processor, Figure 7 shows that there can be an additional cost. Since the application of the preconditioner contains an operation on the cross-point system but the formation of the matrix-vector product does not, it is impossible statically to equally distribute the computational load across all of the processors. The amount of imbalance depends on details of the algorithm and should be considered in chosing an implementation strategy.

5. Domain Decomposition on Networks. Now that we have a description of domain decomposition as appropriate for computers with two-level memory hierarchies, we can look at other computer architectures that might be appropriate for domain decomposition algorithms. An obvious candidate is a network of workstations. A network of 50 workstations can have significant computing power (at 4 megaflops each, such a network has an aggregate power of 200 megaflops), but, more important, such a network has a very large amount of physical memory. For example, with a mere 16 megabytes of memory per workstation, a 50-workstation network will have 800 megabytes of physical memory. Thus, a modest-sized network of workstations has enough memory and computational power to attack significant problems.

This is a good point to raise another issue. Why not use a single workstation and exploit virtual memory? (Another version of this question is, Why not let parallelizing compilers figure out how to organize the algorithm?) The answer is that page thrashing reduces effective computation rate. This is illustrated in Figure 8, where the computation rates for a simple calculation on a workstation are shown. Knowledge of this effect (and

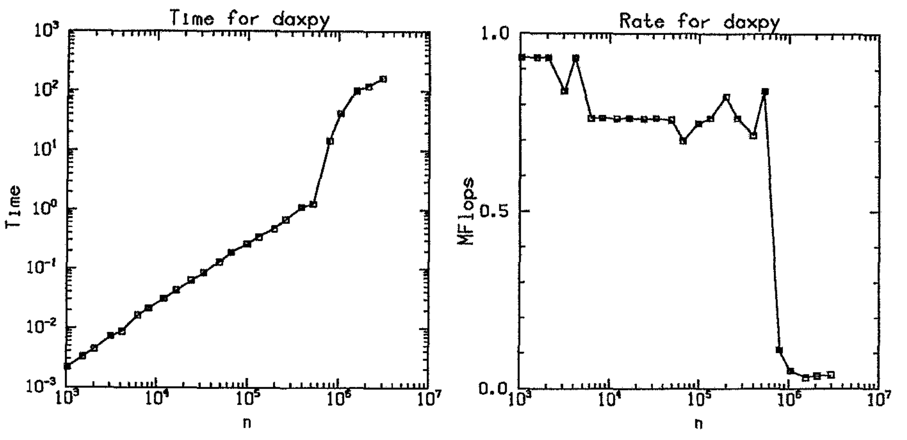

FIG. 8. *Computation rates for a DAXPY operation as a function of vector length on a Sun SPARCStation 1.*

effects related to cache memory) is important; it explains so-called superlinear speedup that is sometimes observed. An example is presented below.

The cost of communicating between processors in a network of workstations can be modeled just as the distributed-memory parallel computers were above. The only difference is that the parameters s and r will probably be somewhat larger. To see how this communication cost affects the performance of domain decomposition algorithms on a network, consider a three-dimensional problem on an $n \times n \times n$ mesh and the cost of computation of the matrix-vector product.

If we assume that the subproblems (each domain) fit in physical memory, the time to compute one iteration or step of the problem is

$$2(s + rn^2) + \frac{n^3}{p},$$

where the domain has been divided into $n \times n \times n/p$ slabs. The speedup is

$$S_p = \frac{n^3}{2(s + rn^2) + \frac{n^3}{p}},$$

so $\partial S_p/\partial p > 0$. Thus, adding processors improves the performance. The one special case is $p = 2$; here, as long as

$$2(s + rn^2) < \frac{n^3}{p},$$

the parallel version will be faster.

There is a more important effect that is related to the discussion of "page thrashing" above. Let the problem be so large that it does not fit in the physical memory of a single processor. Then the time on a single processor may be modeled as

$$n^3\alpha,$$

where $\alpha > 1$ represents the scaling of processor speed when a problem does not fit in memory (α is about 20 for the computation in Figure 8). Let the parallel version of the algorithm use enough processors so that the problem fits within the physical memory of the ensemble of processors. Then, for $n = 100$ (16-Mbyte workstation), the speedup is

$$\begin{aligned} S_p &= \frac{n^3\alpha}{2(s + rn^2) + \frac{n^3}{p}} \\ &= \frac{\alpha p}{2p(s + r10^4)/10^6 + 1} \\ &= \frac{\alpha p}{2 \times 10^{-6}(s + r10^4) + 1.} \end{aligned}$$

Even for large s and r, this is nearly αp, "superlinear" speedup.

This is one of a few situations where small degrees of parallelism are interesting—fitting a problem into memory that would not fit before. Otherwise it is better to use a single processor and wait a little longer.

6. DD and Block Methods. Block methods are methods that divide a problem into blocks and process one block at a time. Such approaches are important in getting the maximum performance out of many vector and matrix operations, including the solution of dense systems of linear equations. These methods do not reduce the actual number of floating-point operations that are used (in some cases, there are actually more operations performed). Instead, they reduce the number of times a data item is read from memory. Block methods are usually organized with a single level of blocks; the block sizes are chosen to match the fast memory of the target computer (cache or vector registers).

Thus, block methods can be considered a form of domain decomposition that has no special treatment of interfaces or global problems. Further, some of the programming tools and methods that have been developed for block methods may be applicable to more general domain decomposition methods. In particular, domain decomposition methods can take advantage of very efficient block method routines to perform local operations such as matrix-vector product and solution of "interior" problems. Other programming tools (such as array sections) can be used to simplify the expression of domain decomposition algorithms.

7. Conclusions. In this paper we have looked at parallel computing applied to domain decomposition algorithms. The keys points to remember are that

- domain decomposition reflects computer hardware (memory hierarchy);
- since perfect parallelism is impossible, speedup can be a misleading measure of effectiveness; and
- time (and space!) complexities may be easily estimated.

As an example of these points, an analysis of the cost of a global cross-point solver suggests that even where the cross-point problem could be computed in parallel with other work, it may be less efficient to do so.

The analysis here also suggests a number of future research areas. Focusing on minimizing computer memory use suggests that single-precision preconditioners may be valuable. Some new RISC processors already have single precision performance that is as much as double the double-precision performance; domain decomposition methods may permit the use of this hardware with the preconditioner without loosing accuracy in the solution.

The highest-performance parallel computers of the future are likely to have more than two important levels of memory hierarchy. These may include cache (or vector registers), local memory, off-processor memory, and mass-storage memory (such as high-speed disks). Domain decomposition techniques may be used to make best use of this structure.

Domain decomposition is appropriate for both SIMD parallel computers and for networks of workstations. In particular, the large penalties for interprocessor communication on networks of workstations represent an extreme case of the two-level memory structure for which domain decomposition is so suited.

REFERENCES

[1] P. E. Bjorstad, J. Braekhus, and A. Hvidsten, *Parallel substructuring algorithms in structural analysis, direct and iterative methods*, in Fourth International Symposium on Domain Decomposition Methods, R. Glowinski, Y. A. Kuznetsov, G. Meurant, J. Périaux, and O. B. Widlund, eds., Philadelphia, 1991, SIAM, pp. 321–340.

[2] T. Chan and R. Schreiber, *Parallel networks for multigrid algorithms: Architecture and complexity*, SIAM J. Sci. Statist. Comput., 6 (1985), pp. 698–711.

[3] L. Cowsar and M. F. Wheeler, *Parallel domain decomposition method for mixed finite elements for elliptic partial differential equations*, in Fourth International Symposium on Domain Decomposition Methods, R. Glowinski, Y. A. Kuznetsov, G. Meurant, J. Périaux, and O. B. Widlund, eds., Philadelphia, 1991, SIAM, pp. 358–372.

[4] J. Dongarra and L. Johnsson, *Solving banded systems on a parallel processor*, Parallel Computing, 5 (1987), pp. 219–246.

[5] M. Dryja and O. B. Widlund, *An additive variant of the Schwarz alternating method for the case of many subregions*, Tech. Rep. TR 339, NYU, Courant Institute, December 1987.

[6] W. D. Gropp and D. E. Keyes, *Complexity of parallel implementation of domain decomposition techniques for elliptic partial differential equations*, SIAM J. Sci. Statist. Comput., 9 (1988), pp. 312–326.

[7] ———, *Domain decomposition on parallel computers*, Impact Comput. Sci. Eng., 1 (1989), pp. 421–439.

[8] ———, *Domain decomposition on parallel computers*, in Second International Symposium on Domain Decomposition Methods, T. F. Chan, R. Glowinski, J. Périaux, and O. Widlund, eds., Philadelphia, 1989, SIAM, pp. 260–268.

[9] ———, *Parallel domain decomposition and the solution of nonlinear systems of equations*, Mathematics and Computer Science Preprint MCS–P186–1090, Argonne National Laboratory, October 1990.

[10] ———, *Parallel performance of domain-decomposed preconditioned Krylov methods for PDEs with adaptive refinement*, MCS-P147-0490, Argonne National Laboratory, Mathematics and Computer Science Division, May 1990. To appear in to SIAM J. Sci. Stat. Comp. as Parallel Performance of Domain-decomposed Preconditioned Krylov Methods for PDEs with Locally Uniform Refinement.

[11] M. Haghoo and W. Proskurowski, *Parallel efficiency of a domain decomposition method*, in Second International Symposium on Domain Decomposition Methods, T. F. Chan, R. Glowinski, J. Périaux, and O. Widlund, eds., Philadelphia, 1989, SIAM, pp. 269–281.

[12] L. Johnsson, *Solving narrow banded systems on ensemble architectures*, ACM Trans. Math. Softw., 11 (1985), pp. 271–288.

[13] H. Jordan, *Interpreting parallel processor performance measurements*, SIAM J. Sci. Statist. Comput., 8 (1987), pp. s220–s226.

[14] D. E. Keyes, *Domain decomposition methods for the parallel computation of reacting flows*, Comput. Phys. Comp., 53 (1989), pp. 181–200.

[15] D. E. Keyes and W. D. Gropp, *A comparison of domain decomposition techniques for elliptic partial differential equations and their parallel implementation*, SIAM J. Sci. Stat. Comp., 8 (1987), pp. s166–s202.

[16] ———, *Domain decomposition techniques for the parallel solution of nonsymmetric systems of elliptic boundary value problems*, Appl. Num. Math., 6 (1990), pp. 281–301.

[17] P. H. Worley, *Information requirements and the implications for parallel computation*, Tech. Rep. STAN-CS-88-1212, Computer Science Department, Stanford University, June 1988.