NASA CR-172,516

NASA Contractor Report 172516

ICASE REPORT NO. 85-2

NASA-CR-172516
19850010318

ICASE

WHERE ARE THE PARALLEL ALGORITHMS?

Robert G. Voigt

LIBRARY COPY

FEB 21 1985

LANGLEY RESEARCH CENTER
LIBRARY, NASA
HAMPTON, VIRGINIA

Contract Nos. NAS1-17070, NAS1-17130

January 1985

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING
NASA Langley Research Center, Hampton, Virginia 23665

Operated by the Universities Space Research Association

NASA

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665

WHERE ARE THE PARALLEL ALGORITHMS?

Robert G. Voigt

Institute for Computer Applications in Science and Engineering

Abstract

Four paradigms that can be useful in developing parallel algorithms are discussed. These include computational complexity analysis, changing the order of computation, asynchronous computation, and divide and conquer. Each is illustrated with an example from scientific computation, and it is shown that computational complexity must be used with great care or an inefficient algorithm may be selected.

Submitted to the 1985 National Computer Conference for publication in the AFIPS NCC Proceedings

---

Research was supported by the National Aeronautics and Space Administration under NASA Contract Nos. NAS1-17070 and NAS1-17130 while the author was in residence at ICASE, NASA Langley Research Center, Hampton, VA 23665.

N85-18627 #

INTRODUCTION

Parallelism has become a major contributor to increased performance in recent years, and it is now accepted that future supercomputers will involve many processors working together in parallel on a single problem. This trend has been brought about by fundamental limits on circuit switching and signal propagation times that inhibit dramatic increases in uniprocessor speeds; it has not been fueled by software developments that might exploit parallelism. In fact, algorithm, programming language, and operating system development lag the pace of hardware advances. In this brief note we will focus on parallelism in algorithm development emphasizing floating point intensive computations that arise in scientific computing.

There have been a number of parallel computers developed in recent years, but the majority of these have been of SIMD type; that is, a single instruction is applied simultaneously to a collection of operands. Even though these machines are architecturally different from vector machines, from an algorithmic point of view they are quite similar because it is natural to view an SIMD machine as executing instructions with vectors as operands.

In spite of some earlier interesting research computers at universities, we are just beginning to see machines of MIMD type where each processor may execute its own independent instruction stream in an asynchronous fashion. Unfortunately, few algorithms have been developed to take advantage of this potentially powerful form of computing. Algorithm development has been motivated by vector computers, and to a lesser extent by SIMD parallel arrays because of their widespread availability.[1]

We will discuss four techniques that can be used as guidelines in the development of parallel algorithms. The first of these, computational

complexity, will be treated in the next section where we will show that if it is not used with great care it can lead to the selection of the wrong algorithm.

Three paradigms that can be useful in the development of parallel algorithms include changing the order of computation, computing asynchronously, and applying the divide-and-conquer concept. These will be illustrated in successive sections.

COMPUTATIONAL COMPLEXITY

Traditionally, one of the most important tools for evaluating algorithms has been computational complexity analysis; that is, operation counts. The fact that the fast Fourier transform of n samples requires $O(n\log n)$ arithmetic operations (here and throughout, log denotes $\log_2$) while the straightforward approach requires $O(n^2)$ provides a clear choice of algorithms for serial computers. With the advent of vector computers such as the Cray 1 and the Cyber 205 with their pipelined arithmetic units, computational complexity remained important because every operation costs some unit of time even it is part of a vector operation. Thus for vectors of length n, an algorithm that requires $\log n$ vector operations will not be faster for sufficiently large n than an algorithm that requires n scalar operations since $n\log n$ operations must be performed. This preservation of arithmetic complexity was made precise by the concept of consistency;[2] an algorithm is said to be consistent if its arithmetic complexity is the same order of magnitude as that for the best serial algorithm.

Unfortunately, computational complexity and consistency do not take into account two important aspects of parallel computation. First, parallel computers can support extra computation at no extra cost if the computation can be organized properly. Second, parallel computers are subject to new overhead costs required, for example, by communication and synchronization that are not reflected by computational complexity. We will now illustrate the importance of these concepts by considering algorithms for the solution of the tridiagonal system of equations $Ax = b$.

If we consider an LU factorization of the matrix A where L is unit lower bidiagonal and U is upper bidiagonal, the usual algorithm is inherently serial. Defining the ith row of these matrices as $(0,\ldots,0,c_i,a_i,b_i,0,\ldots,0)$, $(0,\ldots,0,\ell_i,1,0,\ldots,0)$, and $(0,\ldots,0,u_i,b_i,0,\ldots,0)$ respectively, the ith element of the diagonal of U is given by

$$u_i = a_i - c_i\, b_{i-1}/u_{i-1}; \tag{1}$$

and $\ell_i = c_i/u_{i-1}$. The solution x is obtained by solving $Ly = b$, followed by $Ux = y$, both of which require recursions similar to (1). The computational complexity of the preferred algorithms is $O(n)$ for an $n\times n$ system.

Unfortunately, since u_i depends on u_{i-1}, expression (1) and the other recurrences cannot be evaluated directly in parallel, and we are forced to consider alternatives. The most popular parallel algorithm is known as odd-even reduction or cyclic reduction.[1] The idea is to eliminate the odd-numbered variables in the even-numbered equations by performing elementary row operations. Thus, if $R(2i)$ represents the $2i^{th}$ row of the tridiagonal matrix, the following operations can be performed in parallel for $i = 1,\ldots,\frac{n-1}{2}$,

assuming n is odd:

$$R(2i) - (c_{2i}/a_{2i-1}) * R(2i-1) - (b_{2i}/a_{2i+1}) * R(2i+1). \qquad (2)$$

After the step indicated by (2) is completed, a reordering again yields a tridiagonal system that is only half as large. Thus, in the case that $n = 2^k-1$, the process may be continued for k steps until only one equation remains; then all of the unknowns are recovered in a back substitution process. It has been shown[2] that cyclic reduction requires O(n) operations and is thus consistent. Because the algorithm is consistent and because expression (2) may be evaluated using vector operations cyclic reduction has become the method of choice for vector computers; however, we will see that this may not be the case for parallel computers.

It has been noted[3] that the elimination step (2) may be applied to every equation, not just the even ones, resulting in an algorithm known as odd-even elimination. The equations are reordered, and the elimination step is applied again. After k steps, for $n = 2^k-1$, a diagonal matrix remains, and the solution may be obtained in one more step without a back substitution process. Because the elimination step is applied to every equation for logn steps, O(nlogn) arithmetic operations are required. Thus, the algorithm is not consistent and is not a competitor on serial or vector computers.

The situation is different on parallel computers. It is possible to organize the computation so that the extra work at each step does not require extra time; thus both odd-even reduction and odd-even elimination require log n steps. However, odd-even reduction requires a back substitution phase involving another log n steps; this phase is not required in odd-even

elimintion. Thus odd-even elimination has been shown to be superior on some parallel computers.[4,5]

Another potential advantage for odd-even elimination is that data movement potentially required by the back substitution phase is unnecessary. This can reduce the communication requirements imposed by some parallel architectures.

Thus, we have seen that good parallel algorithms can be ignored if one relies solely on computational complexity as a guideline. In particular, we must look for ways to perform extra computation in parallel if it will result in a reduction in the number of steps required or in the amount of communication.

ORDER OF COMPUTATION

The concept of changing the order of computation, or of reordering, may be viewed as restructuring the computational domain and/or the sequence of operations in order to increase the percentage of the computation that can be done in parallel. For example, we will see that in solving partial differential equations discretized over a grid the order in which the nodes of the grid are numbered may increase or decrease the parallelism of the algorithm to be used. An analogous example is the reordering of the rows and columns of a matrix to create independent submatrices that may be processed in parallel. In fact we have already seen how reordering a matrix can be beneficial, for the odd-even reduction algorithm depends on reordering the equations between steps in order to preserve the parallelism.

A crucial step in solving a partial differential equation discretized on a grid of points representing the domain of interest is selecting the order in which the points will be processed. For example, if the points in Figure 1 are numbered left to right, top to bottom, the resulting linear system will have no particular structure other than being banded. On serial computers, finding orderings that reduce the bandwidth is important because a smaller bandwidth means fewer arithmetic operations are required to solve the system. Other goals such as numbering the points in order to increase the degree of parallelism may be more important on parallel computers.

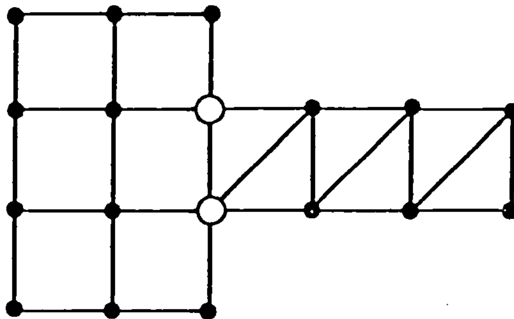

Figure 1. Domain of Points

An ordering known as substructuring[6] has been used by structural engineers in order to decouple structures that are connected by relatively few points. The technique can also be used to introduce parallelism into the system. Conceptually, the situation is depicted in Figure 1, in which the circle points represent interface nodes between the two regions. The nodes in the region may be numbered in any appropriate order, but the interface points are numbered last. This gives rise to a block matrix of the form

$$\begin{bmatrix} A_1 & & C_1 \\ & A_2 & C_2 \\ D_1 & D_2 & B \end{bmatrix}$$

where the A matrices represent the two substructures, the B matrix represents the interface points, and the C and D matrices represent the dependencies between the interface nodes and the two regions. The A matrices may be factored in parallel, and then steps of the form $B - D_i A_i^{-1} C_i$ are used to eliminate the off-diagonal blocks. Finally, the modified B matrix is factored and the solution is obtained in a back substitution process. This technique may be generalized to any number of substructures; the interface nodes must simply separate the structure. However, as the number of substructures increases the size and the complexity of the B matrix also increase providing the algorithm designer with an interesting dilemma. This situation has been studied using a three-dimensional cube as a model.[7] Formulas were obtained to help in the selection of the number of substructures so that the work involved in factoring the modified B matrix will not dominate all other computation.

Although this example involved the direct solution of the linear system, examples of reordering for parallelism exist for iterative methods also.[1] The challenge is to find orderings that increase the degree of parallelism without increasing the arithmetic and communication complexity of some other aspect of the problem.

ASYNCHRONOUS COMPUTATION

Synchronization of computers of MIMD type is used primarily in two situations. In the first, a value such as a sum must be computed from values in some subset of the processors before the computation can continue. In the second, synchronization is used to guarantee a specific order of computation in order to reproduce the behavior of a traditional sequential algorithm.

Depending on the hardware and software of the system, synchronization may be an expensive overhead. For example, several synchronization techniques were studied on the C.mmp computer system, and the cost varied by a factor of 15 with some requiring as much as 30 milliseconds.[8] Another more subtle cost of synchronization is poor processor utilization. Since typically all processors will not reach a synchronization point at the same time, those that arrive before others will be idle until all are ready to proceed.

Thus we would like to consider algorithms that reduce the frequency of synchronization. Fortunately, there are situations in which the synchronization required by the computation of a value dependent on other values distributed throughout the system, or required to mimic sequential behavior, may be eliminated by modifying the algorithm. The Jacobi iterative procedure for approximating the solution of a partial differential equation discretized on a grid requires computing a weighted average of values at neighboring grid points in order to update the approximation at a given grid point. A typical calculation is of the form

$$u_P^{k+1} = \frac{1}{4}\left(u_N^k + u_S^k + u_E^k + u_W^k\right) \tag{3}$$

for the north, south, east, and west neighbors of the point P. Clearly, this algorithm requires synchronization if the u values are being updated by individual processors in a parallel system. However, the algorithm may be modified so that (3) is not forced to use values from the kth iterate. This was the motivation for the pioneering work on chaotic relaxation[9] and for later studies[10]. In its simplest form chaotic or asynchronous iteration can be expressed as

$$u_P^{k+1} = \frac{1}{4}\left(u_N^{k-i_N+1} + u_S^{k-i_S+1} + u_E^{k-i_E+1} + u_W^{k-i_W+1}\right) \tag{4}$$

where i_N, i_S, i_E, and i_W are non-negative integers that may vary with k and P. In words, the algorithm suggested by (4) would have each processor use whatever values were available to compute the next value of the iterate at a given point regardless of which iterate those values were from. Obviously more sophisticated iterative schemes may be adapted to this form of computation.

The properties of asynchronous iterative methods are not well understood. Some theoretical work indicates that the methods will converge under conditions that guarantee the convergence of the corresponding sequential method if the values used on the right side of (4) are from new iterates sufficiently often.[9,10] Unfortunately, as with most iterative methods, the convergence results are asymptotic and do not provide much insight on the observed rate of convergence. Some experimental studies on C.mmp comparing the performance of various asynchronous methods with some sequential ones indicate that the asynchronous methods perform well;[10] however, the sequential methods chosen were not among the best available.

In addition to performance, two aspects of asynchronous iterative methods that require further study include convergence criteria and debugging techniques. Traditional convergence criteria require the computation of a value involving the sum of all solution approximations from the same iterate; at best this requires periodic synchronization, and at worst it may not be possible because different approximations may be on dramatically different iterates. The difficulty with debugging asynchronous programs is that the values produced by the program may not be reproducible because the order of computation may change. This makes isolating errors very difficult.

DIVIDE AND CONQUER

The divide-and-conquer paradigm involves breaking a problem up into smaller subproblems which may be treated independently. Frequently, the degree of independence is a measure of the effectiveness of the algorithm, for it determines the amount and frequency of communication and synchronization.

It is very natural to apply the divide-and-conquer idea to the solution of differential equations by iterative methods. In Figure 1 the region could be divided between two processors with the squares in one and triangles in the other. An algorithm is executed in each processor independently, but periodically information contained at the interface points indicated by the circles must be communicated. Depending on the algorithm, synchronization may be required, but the techniques discussed in the previous section may be used so that the algorithms in the separate processors use whatever data are available rather than waiting for synchronization.

Clearly this idea may be extended to large two- and three-dimensional regions. One of the advantages the technique provides is that, in general, the region may be subdivided to fit the number of processors.

Use of the paradigm also creates opportunities to balance the communication that is required among the various subpieces of the problem. For example, if the processor responsible for the square pieces of the region in Figure 1 updates values of the solution in a left-to-right, top-to-bottom order, then the values at the circled points will be available at different times. On some computer systems the communication of these values could be overlapped with computation. On the other hand, if the order of computation is top-to-bottom, left-to-right, then the values at the circled points will be available at essentially the same time but not until the end of the computation. It would be much more difficult to overlap the required communication on most computer systems.

REFERENCES

1. Ortega, J. and R. Voigt. "Solution of Partial Differential Equations on Vector and Parallel Computers." NASA CR 172500, ICASE Report No. 85-1

2. Lambiotte, J. Jr. and R. Voigt. "The Solution of Tridiagonal Linear Systems on the CDC STAR-100 Computer." ACM Transactions on Mathematical Software, 1 (1975), pp. 308-329.

3. Heller, D. "Some Aspects of the Cylic Reduction Algorithm for Block Tridiagonal Linear Systems." SIAM Journal on Numerical Analysis, 13 (1976), pp. 484-496.

4. Gannon, D. and J. Panetta. "Restructuring SIMPLE for the CHiP Architecture." Parallel Computing, 2 (1985), to appear.

5. Kapur, R. and J. Browne. "Techniques for Solving Block Tridiagonal Linear Systems on Reconfigurable Array Computers." SIAM Journal on Scientific and Statistical Computing, 5 (1984), pp. 701-719.

6. Noor, A., H. Kamel, and R. Fulton. "Substructuring Techniques - Status and Projections." Computers and Structures, 8 (1978), pp. 621-632.

7. Adams, L. and R. Voigt. "A Methodology for Exploiting Parallelism in the Finite Element Process." In J. Kowalik (ed.), High Speed Computation. Berlin: Springer-Verlag, 1984.

8. Oleinick, P. and S. Fuller. "The Implementation of a Parallel Algorithm on C.mmp." Department of Computer Science Report No. CMU-CS-78-125, Carnegie-Mellon University, 1978.

9. Chazan, D. and W. Miranker. "Chaotic Relaxation." Journal of Linear Algebra and Its Applications, 2 (1969), pp. 199-222.

10. Baudet, G. "Asynchronous Iterative Methods for Multiprocessors." Journal of ACM, 25 (1978), pp. 226-244.

1. Report No. NASA CR-172516 ICASE Report No. 85-2	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle WHERE ARE THE PARALLEL ALGORITHMS?		5. Report Date January 1985
		6. Performing Organization Code
7. Author(s) Robert G. Voigt		8. Performing Organization Report No. 85-2
9. Performing Organization Name and Address Institute for Computer Applications in Science and Engineering Mail Stop 132C, NASA Langley Research Center Hampton, VA 23665		10. Work Unit No.
		11. Contract or Grant No. NAS1-17070, NAS1-17130
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Washington, D.C. 20546		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code 505-31-83-01
15. Supplementary Notes Langley Technical Monitor: J. C. South, Jr. Final Report		

16. Abstract

Four paradigms that can be useful in developing parallel algorithms are discussed. These include computational complexity analysis, changing the order of computation, asynchronous computation, and divide and conquer. Each is illustrated with an example from scientific computation, and it is shown that computational complexity must be used with great care or an inefficient algorithm may be selected.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
parameter processing, computational complexity, cyclic reduction, asynchronous iteration	59 - Mathematics in Computer Science 64 - Numerical Analysis Unclassified - Unlimited

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	15	A02

For sale by the National Technical Information Service, Springfield, Virginia 22161

NASA-Langley, 1985